VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Platformy pro paralelní zpracování úloh na GPU

Platforms for Parallel Processing of Task on GPU

2014

Lukáš Lindovský

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Lukáš Lindovský**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Platformy pro paralelní zpracování úloh na GPU
Platforms for Parallel Processing of Task on GPU

Zásady pro vypracování:

V současnosti je i běžný počítač vybaven grafickou kartou (GPU), která je často využívána pouze pro zpracování 2D nebo 3D obrazu, ačkoli nabízí v některých případech větší možnosti paralelizace než mikroprocesory, a jejich výkon tak zůstává nevyužit. Z toho důvodu vznikají nové platformy pro vývoj paralelních algoritmů na grafických kartách.

1. Popište výhody paralelního zpracování úloh na GPU.
2. Popište platformy pro zpracování úloh na GPU, uveďte jejich základní principy a vlastnosti.
3. Pro zvolené úlohy porovnejte výkon těchto platforem, zaměřte se na úlohy z oblasti zpracování dat.
4. Zhodnoťte výhody a nevýhody porovnávaných platforem.

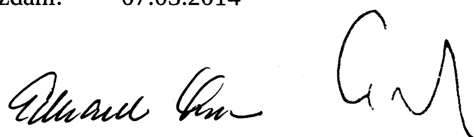
Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Pavel Bednář**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2014 ...Lindovský...

Rád bych na tomto místě poděkoval Ing. Pavlu Bednářovi za odbornou pomoc a vedení při zpracování této bakalářské práce.

Abstrakt

Tato bakalářská práce se zabývá zpracováním úloh na grafické kartě. Konkrétním typem úloh jsou paralelní třídící algoritmy. V první části práce se vyskytuje popis technologií CUDA a OpenCL, ve kterých je později třídící algoritmus implementován. Dále je rozebrán princip daného algoritmu a jeho implementace. Následuje profilování a optimalizace třídícího algoritmu. V poslední částí je testování algoritmů na různých grafických kartách a porovnání obou technologií.

Klíčová slova: CUDA, OpenCL, Odd-Even Merge Sort, Grafická karta, Třídění

Abstract

This thesis deals with the processing tasks to the graphics card. Specific types of tasks are selected sorting algorithms. The first part includes description CUDA and OpenCL technology in which sorting algorithm is implemented. Next it is described the principle of the algorithm and its implementation. Next step is profiling and optimization of sorting algorithm. The last part includes testing these algorithms on different graphics cards and a comparison of both technologies.

Keywords: CUDA, OpenCL, Odd-Even Merge Sort, Graphics Card, Sorting

Seznam použitých zkratek a symbolů

CPU	–	Central Processing Unit
CUDA	–	Compute Unified Device Architecture
GPU	–	Graphic Processing Units
OpenCL	–	Open Computing Language
SM	–	Streaming Multiprocessors
SPMD	–	Single Program, Multiple Data
SIMD	–	Single Instruction, Multiple data

Obsah

1 Úvod 4

2 Technologie CUDA 5
2.1 Porovnání GPU a CPU . . . 5
2.2 Architektura technologie CUDA . . . 6
2.2.1 Programátorské rozdělení . . . 6
2.2.2 Hardwarové rozdělení . . . 7
2.2.3 Typy pamětí . . . 8

3 Technologie OpenCL 10
3.1 OpenCL architektura . . . 10
3.1.1 Platformový model . . . 10
3.1.2 Vykonávací model . . . 11
3.1.3 Paměťový model . . . 13
3.1.4 Programovací model . . . 14
3.2 OpenCL 2.0 . . . 14

4 Třídění 16
4.1 Odd-Even Merge Sort . . . 16
4.1.1 Popis algoritmu . . . 16
4.1.2 Princip implementace Odd-Even merge sort . . . 17
4.1.3 Princip 1. kernelu . . . 17
4.1.4 Princip 2. kernelu . . . 18

5 Testování výkonu 19
5.1 Testované grafické karty . . . 19
5.2 Měření výchozího algoritmu . . . 19
5.3 Profilování 1.fáze algoritmu . . . 20
5.3.1 Profilování výchozího algoritmu . . . 20
5.3.2 Profilování upraveného algoritmu . . . 23
5.4 Profilování 2. fáze algoritmu . . . 25
5.5 Zhodnocení profilování . . . 28

6 Měření a porovnávání 29
6.1 Porovnání výchozího a upraveného algoritmu . . . 29
6.2 Porovnání grafických karet . . . 32
6.3 Měření algoritmu bez sdílené paměti . . . 34

7 Porovnání OpenCL a CUDA 36
7.1 Měření kernelů . . . 36
7.2 Měření trvání celé aplikace . . . 38
7.3 Rozdíly v implementaci . . . 39
7.4 Zhodnocení rozdílů . . . 41

8 Závěr **42**

9 Reference **43**

Přílohy **44**

A Obsah CD **45**
A.1 Technická zpráva 45
A.2 Zdrojové kódy 45

Seznam obrázků

1 Architekruta GPU a CPU 5
2 Šířka paměťového pásma CPU a GPU 6
3 CUDA příklad mřížky 7
4 Automatická škálovatelnost 8
5 CUDA paměťový model 9
6 OpenCL platformový model 11
7 OpenCL příklad mřížky 12
8 OpenCL paměťový model 13
9 Odd-Even Merge Sort pro 8 prvků 17
10 Třídění se sdílenou pamětí 18
11 Paralelní spojování polí 18
12 Spouštění 1. kernelu se sdílenou pamětí - výchozí algoritmus 20
13 Využití karty GeForce GT 630 u výchozího algoritmu. 20
14 Obsazenost multiprocesoru na GeForce GT 630 21
15 Analýza latence - GeForce GT 630 21
16 Analýza latence - GeForce GTX 690 22
17 Využití karty Tesla K20XM 22
18 Využití karty GeForce GT 630 u upraveného algoritmu. 23
19 Problém s propustností paměti u GeForce GTX 690 24
20 Využití karty Tesla K20XM u upraveného algoritmu. 24
21 Spouštění upraveného kernelu se sdílenou pamětí 25
22 Využití karty GeForce GT 630 u 2.fáze algoritmu 25
23 Nabízené možnosti velikosti vláken 26
24 Graf zápisu a čtení 26
25 Využití karty GeForce GTX 690 u 2.fáze algoritmu 27
26 Ukázka závislostí mezi spuštěním 28
27 Porovnání algoritmů na platformě CUDA 31
28 Využití upraveného algoritmu na Tesla K20XM pro 33 554 432 čísel 31
29 Paměťová režie Tesla K20XM pro 131 072 čísel 31
30 Paměťová režie Tesla K20XM pro 33 554 432 čísel 32
31 Porovnání karet - CUDA 33
32 Bližší pohled srovnání GeForce GTX690 a Tesla K20XM 33
33 Porovnání algoritmů se sdílenou pamětí a bez ní. 35
34 Srovnání CUDA a OpenCL na kartě GeForce GT 630 37
35 Srovnání CUDA a OpenCL na kartě GeForce GTX 690 37
36 Doba trvání aplikací - CUDA a OpenCL. 38

1 Úvod

Ještě před několika lety grafické karty sloužili primárně k vykreslení počítačové grafiky. V posledních letech však došlo u grafických karet k obrovskému pokroku ve vývoji, co se týče jak ovladatelnosti, tak výpočetního výkonu. Ty dnešní pak výrazně převyšují vícejádrové procesory nejen svým aritmetickým výkonem, ale také svou paměťovou propustností. Nyní jsou současné architektury grafických karet schopny vykonávat mnohem složitější algoritmy, než tomu bylo několik let zpět a jsou tedy právem označovány jako skutečně masívně paralelní GPU akcelerátory.

Výpočty prováděné grafickými kartami byly však stále limitovány nutností poměrně podrobných znalostí hardwarové architektury, a také nestandardním programováním prostřednictvím vektorů, textur a stínovacích jednotek. Až v roce 2006 uvedla společnost NVIDIA novou architekturu svých GPU čipů, které byly více přizpůsobeny výpočetním úlohám. Zároveň bylo vyvinuto prostředí CUDA, díky kterému je dnes možné programovat aplikace pro grafické karty v běžném jazyce C. O dva roky později vznikla také technologie OpenCL určená pro paralelní výpočty. Ta však není závislá pouze na architektuře grafických karet od výrobce NVIDIA a nemusí být aplikována jen na grafických kartách.

Tato práce se bude zabývat problematikou paralelního třídění čísel. Na každé platformě je třeba mít algoritmy, které efektivně a maximálně využijí dostupné prostředky. Cílem této práce je optimalizovat algoritmus na dané platformě, aby maximálně využíval potenciál grafické karty. Taktéž pomocí tohoto algoritmu porovnat již zmíněné platformy OpenCL a CUDA mezi sebou. Výsledkem by mělo být určení slabších či silnějších míst každé z těchto platforem. Testování bude probíhat na několika grafických kartách, což však díky jejich různým technickým parametrům přináší nutnost upravení algoritmu tak, aby byla grafická karta při paralelním výpočtu maximálně využita a nebyl příliš ztrácen její možný výkon.

V kapitole 2 je rozebrána technologie CUDA a její balík SDK, popis rozdělení této technologie a popis rozdílu mezi GPU a CPU.

Kapitola 3 slouží jako seznámení do technologie OpenCL. Popisuje rozdělení této architektury do jednotlivých čtyř modelů, jejich popis, rozdělení pamětí teto platformy. Závěr kapitoly je ve zkratce věnován nové verzi OpenCL.

V kapitole 4 jsou blíže popsány implementace jednotlivých třídících algoritmů v jazyce CUDA. Kapitola 5 se věnuje analýze algoritmů na různých grafických kartách a jejich následné optimalizaci pro zlepšení výkonu.

Další kapitola 6 je věnována samotnému testování, ve které je měření časů, porovnání implementace bez sdílené paměti a srovnávání různých grafických karet.

Předposlední kapitola 7 se věnuje porovnání obou technologií a popsání jejich hlavních rozdílů.

Závěrečná kapitola 8 shrnuje dosažené výsledky práce.

2 Technologie CUDA

CUDA je technologie umožňující paralelní zpracování úloh implementovaných na jádrech grafické karty. Tato architektura je dostupná pouze na grafických akcelerátorech od společnosti NVIDIA, která ji vyvíjí. Společnost NVIDIA uvolnila první verzi CUDA SDK 1.0 v roce 2007 pro grafické karty založené na architektuře G 80. Nyní je nejnovější verze 5.5, která vyšla v polovině roku 2013. Chystá se verze 6.0, která již má hotovou specifikaci pro registrované vývojáře.

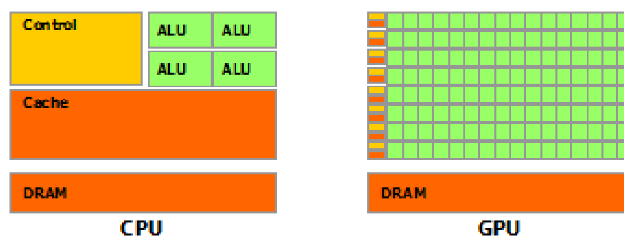
Celý tento balík obsahuje:

- Daný ovládač pro grafickou kartu NVIDIA. Pro případné testování není tato grafická karta nutná. Balík umožňuje simulaci tohoto zařízení pro ověření správnosti a funkčnosti daných výpočtů, ale bez urychlení, jelikož tyto výpočty simuluje procesor.
- CUDA Toolkit, který obsahuje překladač nvcc, hlavičkové soubory a knihovny.
- CUDA SDK, kde se jedná o názorné příklady různých algoritmů a dokumentaci.

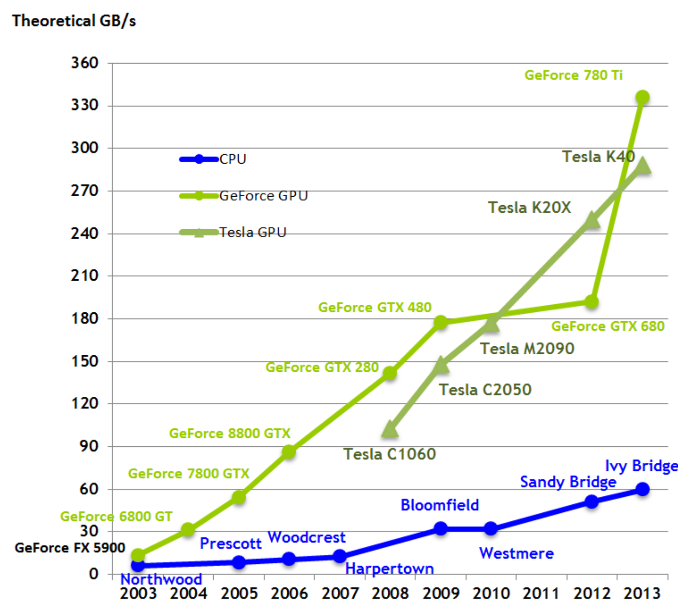
2.1 Porovnání GPU a CPU

Současná myšlenka paralelismu na GPU spočívá v použití CPU ve spolupráci s GPU, tak že sekvenční část je vykonávána na CPU, a výpočetně náročná část bude zpracovávána paralelně pomocí GPU. GPU architektura umožňuje oproti CPU vykonávat paralelně až tisíce vláken. Toto paralelní zpracování výpočtů se v dnešní době stává silnou zbraní při vývoji softwaru.

Polovodičový průmysl se vydal dvěma různými směry. První směr je více-jádrový, ten se snaží udržet vysokou rychlost sekvenčních programů s využitím většího počtu jader, které vykonávají tyto instrukce mimo pořadí. Druhý směr se se přiklání k mnoho-jádrové architektuře vykonávající instrukce v pořadí a navržené s vysokou propustností. V tomto směru se drží vývoj grafických karet, které v sobě nesou více aritmetických jednotek pro výpočet (lze vidět na obrázku 1), má tedy mnoho-jádrovou architekturu, která má o mnoho větší propustnost a vyšší výkon vykonávání výpočtů s plovoucí desetinnou čárkou (znázorněno na obrázku 2) [1].



Obrázek 1: Architekruta GPU a CPU, převzato z [8].



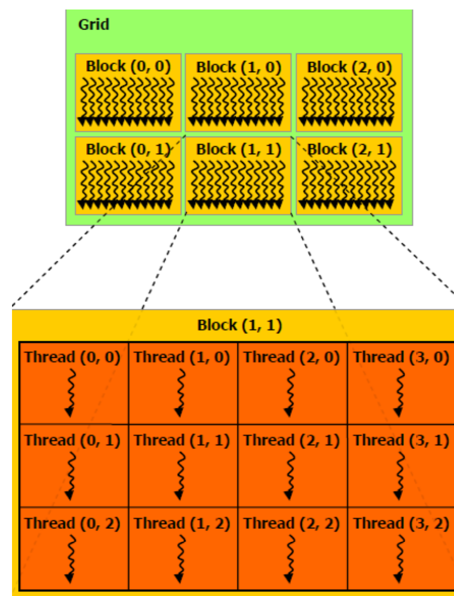
Obrázek 2: Šířka paměťového pásma CPU a GPU, převzato z [8]

2.2 Architektura technologie CUDA

CUDA C/C++ rozšiřuje klasický C/C++ jazyk tím, že umožňuje definovat funkci nazývanou **kernel** (jádro), který je spouštěný paralelně N-krát v N-různých CUDA vláknech. Kernel je identifikován pomocí klíčového slova `__global__`, dále udáním počtu CUDA vláken, ve kterých bude spuštěn. Je zde i možnost definovat CUDA funkce na zařízení, které jsou označené klíčovým slovem `__device__`. Tuto funkci lze volat pouze z jádra, nebo z jiné funkce zařízení. Máme zde i možnost dát před funkci `__host__`, což znamená, že funkce bude brána jako CUDA funkce hosta. Funkce hosta je tradiční funkce jazyka C, která je spouštěna buď na hostovi, nebo případně je volána z jiné funkce hosta. Je vhodné zdůraznit, že ve výchozím nastavení jsou všechny funkce v CUDA nastavené jako funkce hosta [1].

2.2.1 Programátorské rozdělení

Vlákna jsou organizována do 1D, 2D nebo 3D bloků (thread block), každé vlákno má své unikátní ID, které je uložené v zabudované proměnné `threadIdx`. Tyto vlákna jdou v rámci bloku synchronizovat a mohou sdílet data přes sdílenou paměť. Na současných GPU můžeme mít v jednom bloku až 1024 vláken. Bloky jsou organizovány do 1D, 2D nebo 3D mřížky (grid), které lze identifikovat pomocí zabudované proměnné `blockIdx`. Každý blok vláken je schopen pracovat nezávisle na jiném bloku. Počet bloků často nastavujeme v závislosti na datech jaké zpracováváme. Příklad rozložení mřížky lze vidět na obrázku 3.



Obrázek 3: CUDA příklad mřížky, převzato z [14].

2.2.2 Hardwarové rozdělení

Moderní GPU obsahují sadu multiprocesorů, kde každý multiprocesor má řadu skalárních procesorů, registrů, sdílenou paměť, mezipaměť pro konstantní paměť a paměť pro textury. Kromě toho má GPU velké množství globální paměti, která je přístupná z hostitelského procesoru. Multiprocesor vykonává jeden nebo více bloků paralelně a každý obsahuje 32 procesorových jader. Multiprocesor provádí až 32 vláken paralelně, což se nazývá **warp**.

V GPU s větším počtem multiprocesorů se spustí program v kratším čase, než v GPU s menším počtem multiprocesorů. Na obrázku 4 je dobře vidět, jak dochází ke škálovatelnosti. S větším počtem multiprocesorů dochází k rovnoměrnému rozdělení bloků každému z nich.

Obrázek 4: Automatická škálovatelnost, převzato z [8].

2.2.3 Typy pamětí

CUDA podporuje několik druhů pamětí, se kterými můžeme pracovat. Každá z těchto typů paměti má svou určitou jedinečnou vlastnost.

- **Globální paměť** je největší zástupce z pamětí, která je k dispozici pro všechna vlákna bez ohledu na to, kde jsou prováděna. Přístup ke globální paměti je však pomalejší než přístup k lokální paměti, proto by tato operace měla být co nejvíce minimalizována.

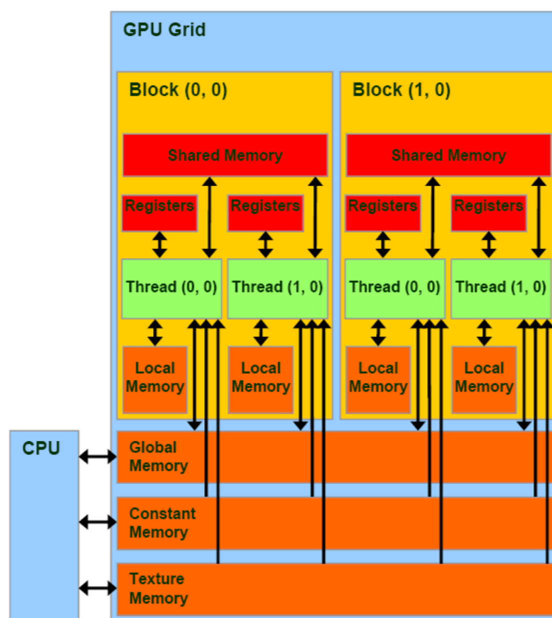
- **Lokální paměť** je poměrně malá paměť, ke které může přistupovat pouze jeden multiprocesor.

- **Sdílená paměť** je umístěná přímo na čipu, její maximální velikost je 48 KB a je přístupná všem vláknům ve stejném bloku. Ačkoli je zde omezení velikosti této paměti, doporučuje se tuto paměť využívat díky její rychlosti. Sdílená paměť je účinným prostředkem ke sdílení vstupních dat a průběžných výsledků práce mezi vlákny v rámci jednoho bloku.

- **Registr vlákna** umožňuje, že každé vlákno má k dispozici svůj registr paměti a jediné k němu má přístup. Čtení a zápis proměnné probíhá velmi vysokou rychlostí a značně paralelním způsobem. Jedná se o další typ paměti umístěný přímo na čipu.

- **Konstantní paměť** slouží pro uložení konstantních proměnných, které jsou platné pro všechny bloky. Konstantní proměnná je alokována po celou dobu běhu aplikace v globální paměti, a přesto přístup k ní je rychlejší díky uložení v mezipaměti. Maximální velikost této paměti činí 64 KB.

- **Paměť textur** je pouze pro čtení, disponuje cache pamětí. Je optimalizována pro 2D prostorovou lokalitu [14].



Obrázek 5: CUDA paměťový model, převzato z [14].

K alokaci potřebné paměti na grafické kartě slouží funkce `cudaMalloc()`, která není pouze podobná funkci `malloc()` z klasické C knihovny, ale je jejím rozšířením. První parametr, který tato funkce potřebuje je adresa ukazatele proměnné, která by měla být přetypována na `(void**)`. Druhý parametr funkce udává velikost objektu, která má být přidělena v bajtech.

Alokace samotná k práci nestačí. Pokud je připraven alokovaný prostor na grafické kartě, je potřeba přenést data z hosta na grafickou kartu. K tomu slouží funkce `cudaMemcpy()`, která přijímá 4 parametry. První parametr je ukazatel na místo určení, kde se mají data nakopírovat a druhý parametr je ukazatel na místo ze kterého se data mají kopírovat. Třetí parametr určuje počet bajtů ke zkopírování. Poslední parametr určuje, ze kterého zařízení budou data kopírována, a kde budou kopírována. Jsou zde čtyři možnosti kopírování dat: z hostitelské paměti na paměť hosta, z paměti hosta na paměť zařízení, z paměti zařízení na paměť zařízení, a z paměti zařízení na paměť hosta [1].

3 Technologie OpenCL

OpenCL je otevřený standard pro paralelní programování a zpracování výpočetních úloh nejen na grafickém akcelerátoru. Poskytuje jednotné programovací rozhraní jak pro software běžící na klasických procesorech, grafických kartách, Cell procesorech, či na jiném zařízení, které může zpracovávat úlohy paralelně. OpenCL 1.0 standart byl vydán skupinou The Khronos Group v prosinci roku 2008. První OpenCL SDK pro programování na grafických kartách se objevilo až v září roku 2009. Nyní je nejnovější verze 2.0, která vyšla v listopadu 2013, a jedná se tedy o velmi mladou verzi tohoto standartu.

3.1 OpenCL architektura

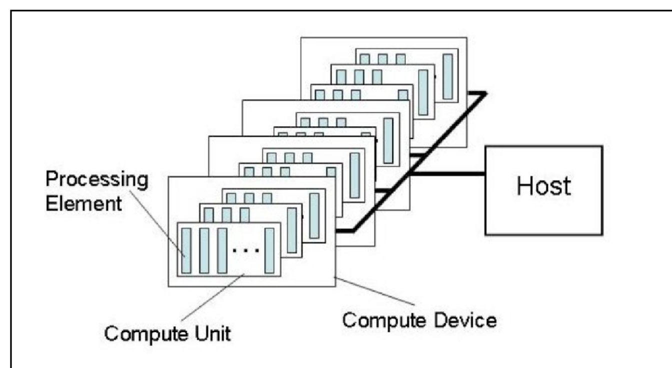
Architekturu OpenCL lze rozdělit do několika hierarchicky uspořádaných modelů:

- Platformový model
- Vykonávací model
- Paměťový model
- Programovací model

3.1.1 Platformový model

OpenCL model platformy popisuje strukturu jednotlivých systémových částí, na kterých poběží software vyvíjený standartem OpenCL.

Definice platformového modelu je, že hostující zařízení (host) je připojeno s jedním nebo více OpenCL zařízení, na kterých jsou proudy instrukcí (kernely) vykonávány. Toto zařízení může být CPU, GPU nebo jiný procesor, který poskytuje hardware podporovaný platformou OpenCL . Obrázek 6 zobrazuje platformový model, který obsahuje jedno host zařízení a navíc více OpenCL kompatibilních výpočetních zařízení (Compute Device). V jednom systému tak může být například kompatibilní procesor společně s grafickou kartou podporující OpenCL. Každé výpočetní zařízení se skládá z několika výpočetních jednotek (Compute Unit). Tyto výpočetní jednotky jsou jádra procesoru nebo multiprocesory grafické karty. Výpočetní jednotky se dále skládají z jednoho nebo více výpočetních elementů (Compute Unit), které vykonávají instrukce jako SIMD (Single Instruction, Multiple data) nebo SPMD (Single Program, Multiple Data). SPMD instrukce jsou typicky prováděny na univerzálních zařízeních, jako jsou procesory, kdežto SIMD instrukce vyžadují vektorový procesor, jako je například GPU nebo vektorová jednotka v procesoru [5].



Obrázek 6: OpenCL platformový model, převzato z [2]

3.1.2 Vykonávací model

Vykonávání OpenCL aplikace se skládá z hostitelského programu a kolekce jednoho nebo více kernelů. Tento model také definuje, jakým způsobem se budou kernely na OpenCL zařízení vykonávat. Kernely jsou spouštěny na OpenCL zařízeních a provádějí skutečnou práci aplikace. Jsou to jednoduché funkce, které zajišťují paralelní zpracování úlohy na OpenCL zařízení [2]. OpenCL má dva typy kernelů:

- **OpenCL kernely**: funkce psané v OpenCL C jazyce a kompilované OpenCL kompilátorem.
- **Nativní kernely**: funkce napsané mimo OpenCL, které jsou přístupné uvnitř OpenCL pomocí ukazatele. Toto mohou být například funkce ze specializované knihovny nebo mohou být definovány ve zdrojovém kódu hostitele [2].

Hostitelský program vydá příkaz, kterým předá kernel pro spuštění na OpenCL zařízení. Poté se vytvoří celočíselný indexovaný prostor. Instance kernelu spuštěná na jednom výpočetním prvku se nazývá pracovní položka (work-item), která je jasně identifikována svými souřadnicemi v indexovaném prostoru.

Celočíselný indexovaný N-rozměrný prostor OpenCL se nazývá NDRange. N může nabývat hodnot 1,2 nebo 3. NDRange je definováno jako celočíselné pole o délce zmíněného N, které specifikuje velikost prostoru pro každý rozměr. Každé globální a lokální ID pracovní položky je N-rozměrnou N-ticí. Globální ID má rozsah od nuly do počtu prvků v daném rozměru mínus jedna. Pracovní položky jsou přiřazeny ke pracovní skupině a označeny lokálním ID v rozsahu od nuly do velikosti pracovní skupiny mínus jedna. Z toho vyplývá, že kombinace ID pracovní skupiny a lokální ID pracovní položky uvnitř pracovní skupiny jasně identifikuje pracovní položku. Pracovní položky jsou organizovány do pracovních skupin (work-groups). Pracovní skupiny mají taktéž své jednoznačné ID, které má stejný rozměr jako ID pracovní položky [2].

Pokud si vezmeme jako příklad dvourozměrné pole 2D rozsahu. Použijeme označení g pro globální ID pracovní položky v každé dimenzi označené dolním indexem písmeny x a y. Označení písmenem G budeme uvažovat velikost indexovaného prostoru v každé

dimenzi. Takže každá pracovní položka má souřadnice (gx,gy) v globálním ND Range indexovaném prostoru (Gx,Gy) a nabírá hodnot [. . . (Gx -1), 0...Gy -1)].

ND Range prostor dělíme do pracovních skupin, které budou mít označení ID písmenem w. Písmeno W bude označení pro počet pracovních skupin v každé dimenzi, ta bude opět označena dolními indexy x a y. Velikost pracovní skupiny v každém směru (x a y v našem případě 2D rozsahu) se používá k definování lokálního ID pro každý pracovní bod. Velikost lokálního ID v každém rozměru (x a y) je označeno písmenem L a lokální ID uvnitř skupiny je označeno l [5].

Každá pracovní skupina je velikosti Lx a Ly, kterou lze získat ze vztahu:

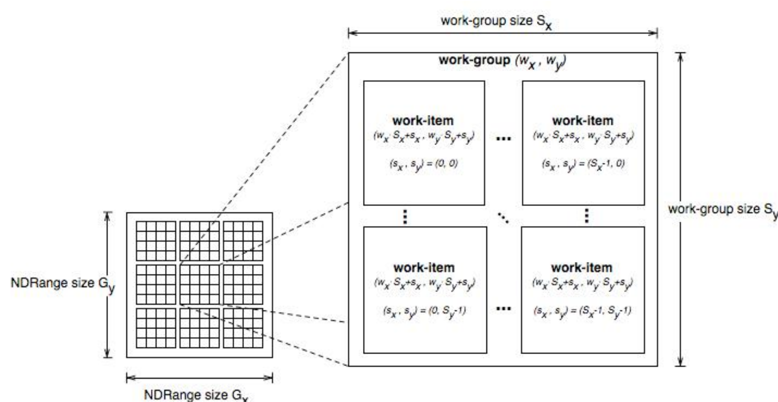
$$Lx = Gx/Wx$$
$$Ly = Gy/Wy$$

Můžeme definovat pracovní položku globálním ID (gx,gx), nebo kombinací lokální ID (lx,ly) a ID pracovní skupiny (wx,wy), platí tedy následující vztah:

$$gx = wx * Lx + lx$$
$$gy = wy * Ly + ly$$

Popřípadě můžeme zjistit lokální ID pracovní položky nebo ID pracovní skupiny následovně:

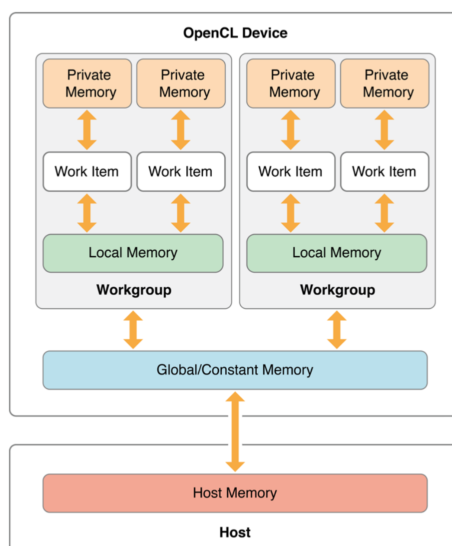
$$wx = gx/Lx$$
$$wy = gy/Ly$$
$$lx = gx\%Lx$$
$$ly = gy\%Ly$$



Obrázek 7: OpenCL příklad mřížky, převzato z [5].

3.1.3 Paměťový model

Paměť v OpenCL se dělí do dvou částí. Ta první je hostitelská paměť, která je viditelná pouze pro hosta. OpenCL definuje pouze to, jak tato paměť spolupracuje s OpenCL objekty a její chování je definováno mimo OpenCL. Druhou pamětí lze chápat jako paměť zařízení, která má k dispozici přístup přímo na kernelech spouštěných na OpenCL zařízení [2].



Obrázek 8: OpenCL paměťový model, převzato z [13]

OpenCL paměťový model definuje čtyři typy pamětí v zařízení:

- **Globální paměť** - Tato paměť umožňuje čtení i zápis pro všechny pracovní položky ve všech pracovních skupinách. Pracovní položky mohou číst nebo zapisovat z jakékoliv části paměťového objektu. Podle schopnosti zařízení může být zápis a čtení do globální paměti cachováno.

- **Konstantní paměť** – Tato část globální paměti zůstává během běhu kernelu konstantní. Hostitelský systém alokuje a inicializuje paměť objektů, které budou v konstantní paměti.

- **Lokální paměť** – Paměť pro pracovní skupiny, která může být využita pro uložení proměnných a sdílení všemi pracovními položkami v této pracovní skupině. Může být implementována jako vyhrazená část OpenCL zařízení. V jiném případě může být také mapována na části globální paměti. Přístup k této paměti je řádově rychlejší než přístup ke globální paměti.

- **Privátní paměť** – Tato paměť je soukromou pamětí pro pracovní položky. Proměnné nadefinované v privátní paměti jedné pracovní položky nejsou viditelné pro ostatní pracovní položky [13].

Hostitelská paměť a paměť zařízení pracují nezávisle na sobě, nicméně spolu potřebují komunikovat a to lze zajistit pomocí mapování a odmapování části paměti nebo explicitního kopírování dat. Pří explicitním kopírování dat se použijí fronty příkazů od hosta, které zajistí přenos dat mezi paměťovým objektem a pamětí hosta. Tato paměť může, ale nemusí být pro přenos příkazů blokována. Pokud je zavolána funkce zablokování, mohou být ostatní přidružené zdroje bezpečně použity znovu. Při neblokování zařadí OpenCL příkaz do fronty bez ohledu na to, zda je paměť hosta ve stavu bezpečného použití. Mapování a odmapování je metoda, která umožňuje hostovi mapovat oblast z paměťového objektu do vlastního adresového prostoru. Jakmile je mapování dokončeno, hostitel může číst nebo zapisovat do této oblasti. Poté host odmapuje oblast, pokud je čtení nebo zápis kompletně dokončen.

3.1.4 Programovací model

OpenCL definuje datově paralelní a úkolově paralelní programovací modely. Existuje také podoba hybridního modelu, což je úkolově paralelní model spojený s datově paralelním modelem.

Datově paralelní programovací model je nejlepší variantou při použití datových struktur, kde mohou být jednotlivé prvky zpracovány současně. To je definováno jako sled instrukcí, které se aplikují současně na elementy datové struktury. Přičemž indexovaní definuje přesné mapování dat na pracovní položky [5].

Úkolově paralelní programovací model je model, ve kterém běží jedna instance kernelu nezávisle na indexovaném prostoru. OpenCL tedy definuje úkol jako kernel, který je vykonáván jako jedna pracovní položka. Jsou tři možnosti, jak se může v tomto modelu vyjádřit paralelismus:

1. Vyjádřením pomocí vektorových operací nad vektorovými typy.
2. Zavedením více úkolů do fronty.
3. Použití nativního kernelu, kde paralelismus je vyjádřen pomocí prostředí mimo OpenCL [5].

3.2 OpenCL 2.0

V listopadu roku 2013 společnost Khronos Group oznámila, že má hotovou specifikaci novou generaci rozhraní OpenCL ve verzi 2.0. Ta přinesla několik revolučních novinek a vylepšení.

- Sdílená virtuální paměť - Kód běžící na GPU si může předávat přímo adresy a ukazatele s kódem běžícím na CPU a naopak.
- Vnořený paralelismus - Umožní vynechání určité komunikace mezi hostem a zařízením. Práci grafickému čipu už nemusí zadávat procesor počítače.
- Generický adresový prostor - Funkce lze nyní psát bez nutnosti specifikace adresových prostorů.

- Obrázky - Vylepšená podpora pro sRGB a 3D obraz. Kernely mohou číst a zapisovat stejný obraz.
- Roury - Jedná se o paměťové objekty, které uchovávají data organizované FIFO systémem. Pro zápis a čtení do rour je k dispozici několik užitečných funkcí.

4 Třídění

Třídění je jedním z nejdůležitějších stavebních prvků ve výpočetní technice. Ačkoli třídění na CPU je poměrně jednoduché a největší problém je zvolit správný třídící algoritmus, tak oproti tomu třídění na GPU přináší o mnoho více nástrah. Výsledný čas často nezáleží pouze na kvalitě algoritmu, ale důležitý faktor hraje také správný přístup do paměti, synchronizace a efektivní využití paralelismu. Je tedy důležité provádět třídění přímo na grafické kartě z důvodu velkých nákladu na zbytečný přenos dat mezi GPU a CPU.

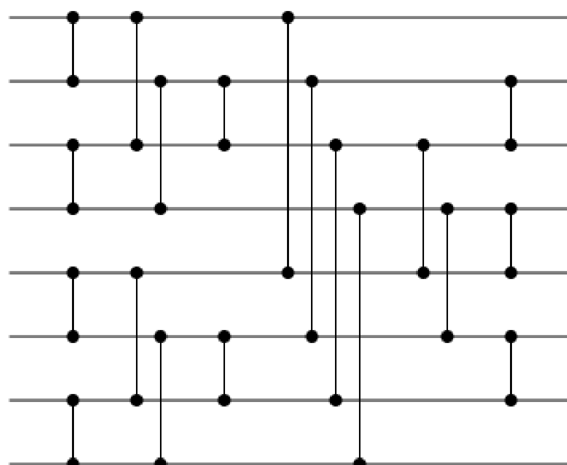
4.1 Odd-Even Merge Sort

Jedná se o třídící algoritmus, který je založený na technikách bubble sortu. Díky své konstrukci je vhodný k paralelnímu běhu. Princip algoritmu spočívá ve sloučení dvou podřízených polí o délce N/2, které jsou uložené v levé a pravé části pole o velikosti N.

4.1.1 Popis algoritmu

Pokud vezmeme v úvahu příklad na obrázku 9, kde velikost posloupnosti je N=8 , jsou jednotlivé prvky jsou zastoupeny vodorovnými úsečkami. Zde jsou vidět jednotlivé fáze setřídění této posloupnosti. Svislá čára mezi jednotlivými prvky znázorňuje, které dva prvky jsou v danou chvíli porovnávány a mohou si tedy navzájem vyměnit své pozice.

Algoritmus začíná od posloupnosti velikosti 2 prvků. V první fázi se porovnávají prvky a_0 s a_1, a_2 s a_3 až a_{n-2} s a_{n-1}, přičemž po dokončení lze říct, že několik posloupností o velikosti 2 prvků je seřazených. V dalším kroku je třeba sloučit dvě setříděné posloupnosti o velikosti 2 prvků do jedné posloupnosti o velikosti 4 prvků. Je třeba porovnat číslo v první setříděné posloupnosti s číslem z druhé setříděné posloupnosti, které má v posloupnosti stejný index jako číslo první. V tomto případě tedy porovnáváme a_0 s a_2, a_1 s a_3. Vznikne tedy posloupnost o 4 prvcích, kde první bude nejnižší z celé posloupnosti a poslední číslo bude nejvyšší číslo z této posloupnosti (pokud uvažujeme vzestupné třídění). Je třeba však ještě provést porovnání jejich sousedních prvků a_2 a a_1, které se nacházejí mezi nimi. Po tomto kroku je setříděná posloupnost o velikosti 4 prvků. Celý tento postup se provádí také pro druhou část výchozího pole. Vzniknout nám dvě setříděné pole o velikostech 4 prvků. Tyto dvě setříděné pole se sloučí dohromady stejným postupem. Takto algoritmus postupuje rekurzivně, dokud není setříděná celá posloupnost.



Obrázek 9: Odd-Even Merge Sort pro 8 prvků, převzato z [15].

4.1.2 Princip implementace Odd-Even merge sort

Celý algoritmus Odd-Even merge sortu se skládá ze dvou fází. Každá tato fáze zahrnuje jeden kernel.

V první fázi se používá kernel, který využívá sdílenou paměť. Princip tohoto kernelu spočívá v tom, že rozdělí vstupní pole na několik menších polí, které setřídí. Tento postup je však omezen maximální velikostí sdílené paměti.

Druhá fáze volá kernel, který využívá pouze globální paměť. Tento kernel již počítá s tím, že pole bylo rozděleno na několik menších polí předchozím kernelem. Tyto menší pole jsou již setříděné. Jeho úkol tedy spočívá v tom, že již setříděné menší pole o velikosti M spojí do jednoho pole, které bude mít velikost 2M. Toto výsledné pole bude setříděné. Takto je tento kernel volán několikrát v cyklu, dokud nejsou všechny menší pole spojeny.

Protože tento kernel bude mít za úkol spojit i dvě pole o velikosti v řádech miliónů čísel, tak nelze využít sdílenou paměť, která má omezenou velikost.

Při třídění 131 072 čísel zabírá 1. kernel pracující se sdílenou pamětí 13,4% výpočtů a 2. kernel pracující s globální pamětí 86,6% výpočtů. Pří třídění 33 554 432 čísel zabírá 1. kernel pracující se sdílenou pamětí 5,5% výpočtů a 2. kernel pracující s globální pamětí 94,5% výpočtu. Jak jde usoudit, s rostoucím počtem tříděných čísel má druhý kernel využívající globální paměť větší podíl na třídění než kernel první.

4.1.3 Princip 1. kernelu

Jak již bylo napsáno, první fáze Odd-Even merge sortu využívá sdílenou paměť, do které vlákna uloží všechna čísla z posloupnosti. Toto je však omezeno velikostí sdílené paměti. Důležitá část algoritmu je znázorněna na obrázku 10. Řádek 1 je hlavní cyklus, který určuje velikost již setříděné posloupnosti. Tato hodnota startuje velikostí 2 prvků a neustále se zdvojnásobuje, dokud není posloupnost zcela setříděna. Řádek 6 nám zajišťuje volání funkce, která jednoduše porovná dva klíče s hodnotami indexu, které dostává jako

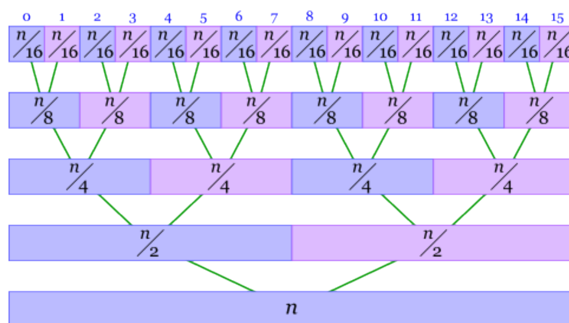
parametry. V závislosti na parametru "dir"tyto klíče buď přehodí, nebo ponechá na svém místě. Vnořený cyklus for na řádku 9 nám zajišťuje závěrečné dotřídění, kde se opět volá funkce, která porovnává dva klíče.

```
1  for (int size = 2; size <= arrayLength; size <<= 1){
2      int stride = size / 2;
3      int offset = threadIdx.x & (stride - 1);
4      __syncthreads();
5      int pos = 2 * threadIdx.x - (threadIdx.x & (stride - 1));
6      Comparator(s_key[pos +0],s_val[pos + 0],s_key[pos + stride], s_val[pos + stride], dir);
7      stride >>= 1;       // stride = stride/2
8
9      for (; stride > 0; stride >>= 1){
10         __syncthreads();
11         int pos = 2 * threadIdx.x - (threadIdx.x & (stride - 1));
12         if (offset >= stride){
13             Comparator(s_key[pos - stride], s_val[pos - stride], s_key[pos +0], s_val[pos +0], dir);
14 }}}
```

Obrázek 10: Třídění se sdílenou pamětí

4.1.4 Princip 2. kernelu

Tento kernel využívá pouze přístupu do globální paměti. Jeho princip je stejný jako v případě prvního kernelu, který využívá sdílenou paměť. Avšak cílem tohoto kernelu je sloučit menší pole, které jsou již setříděné. V této práci je tento kernel implementován tak, že počítá s tím, že mu první kernel již předtřídil menší pole o velikosti 1 024 prvků. Jeho první prací tak je sloučit dvě pole o velikosti 1 024 prvků, tak aby vzniklo setříděné pole o velikosti 2 048 prvků. Takto pokračuje rekurzivně, dokud není celé pole setříděno. Postup je tedy stejný jako je znázorněn na obrázku 9. V tomto případě se však jedná o pole s vysokým počtem prvků, a proto nelze využít sdílenou paměť. Tento paralelní způsob postupného slučování polí je znázorněn na obrázku 11.



Obrázek 11: Paralelní spojování polí, převzato z [16].

5 Testování výkonu

5.1 Testované grafické karty

Pro testování byly k dispozici 3 grafické karty. Všechny tyto karty byly vyrobeny společností NVIDIA. Konkrétně se jednalo o karty GeForce 630, GeForce GTX 690. Měření proběhlo také na školním databázovém serveru zvaným CODD. Tento server disponuje grafickou kartou Tesla K20Xm. Jednotlivé parametry všech těchto karet jsou k dispozici v tabulce 1.

Parametr	**Nvidia GeForce GT 630 (OEM)**	**Nvidia GeForce GTX 690**	**NVIDIA Tesla K20Xm**
Velikost globální paměti	2048 MB	2048 MB	5 760 MB
Počet multiprocesorů	1	8	14
Počet CUDA jader	192	1 536	2 688
Maximální velikost konstantní paměti	65 536 bajtů	65 536 bajtů	65 536 bajtů
Maximální velikost sdílené paměti	49 152 bajtů	49 152 bajtů	49 152 bajtů
Maximální počet registrů na blok	65 536 bajtů	65 536 bajtů	65 536 bajtů
Velikost warpu	32	32	32
Maximální počet vláken na multiprocesor	2048	2048	2048
Maximální počet vláken na blok	1024	1024	1024

Tabulka 1: Parametry testovaných grafických karet

5.2 Měření výchozího algoritmu

První krokem bylo profilování výchozího algoritmu. Při spouštění nevyladěného algoritmu na různých grafických kartách se může stát, že dojde ke ztrátě výkonu. To způsobují zejména různé parametry grafických karet. Testování a ladění výchozího algoritmu bylo věnováno programu napsaném pro platformu CUDA. K tomuto ladění byl nápomocen nástroj Visual Profiler a ladění probíhalo pro třídění 32 768 čísel. Jelikož algoritmus implementovaný na platformě OpenCL má totožnou strukturu jako algoritmus na platformě CUDA, bude se předpokládat, že výsledky profilování pro danou grafickou kartu vycházejí stejně.

Jelikož program je rozdělen na dvě fáze, přičemž každá fáze pracuje s jiným kernelem, bylo potřeba profilovat oba tyto kernely. Každý tento kernel může mít své slabé místo, proto bylo potřeba po profilování toto slabé místo určit a v nejlepším případě algoritmus vylepšit.

5.3 Profilování 1.fáze algoritmu

Tato část bude profilovat 1. fázi algoritmu, čili kernel používající sdílenou paměť. Sdílená paměť uvnitř kernelu je zastoupena dvěma poli o velikosti 1024 prvků. Na obrázku 12 můžeme vidět, že celkové pole bude rozděleno pomocí sdílené paměti na 128 polí o velikosti 256 čísel, kde každé toto pole o velikosti 256 čísel bude ve výsledku setříděno. K tomu jeden blok potřebuje polovinu vláken velikosti jednoho takového pole, čili 128 vláken na blok. Toto rozdělení vychází pro třídění 32 768 čísel.

```
1  int arrayLength = 256;
2  int  blockCount = N / arrayLength;          // 32 768 / 256 = 128
3  int threadCount = arrayLength/2;
4  oddEvenMergeSortShared<<<blockCount, threadCount>>>(d_DstKey, d_DstVal, d_SrcKey,
       d_SrcVal, arrayLength, dir);
```

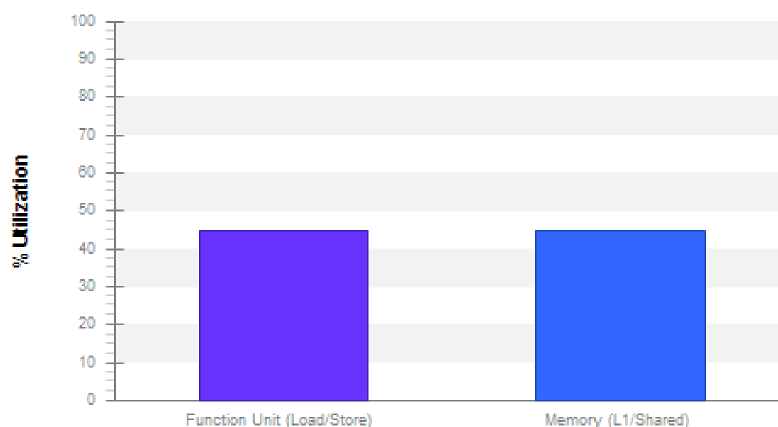
Obrázek 12: Spouštění 1. kernelu se sdílenou pamětí - výchozí algoritmus

5.3.1 Profilování výchozího algoritmu

Jak první byl profilován kernel se sdílenou pamětí a to na všech třech grafických kartách. Výsledky těchto profilování budou rozebrány jednotlivě.

Profilování na GeForce GT 630

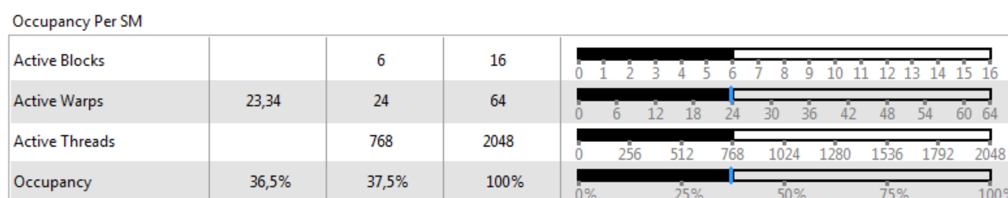
Dosažená obsazenost karty byla pouhých 36,5% a její využití se pohybovalo okolo 45%, jak můžeme vidět na obrázku 13.



Obrázek 13: Využití karty GeForce GT 630 u výchozího algoritmu.

Na obrázku 15 jde vidět, co je hlavní příčinou ztráty výkonu. Problémem bylo špatné nastavení sdílené paměti. K dispozici je celkem 48 KB sdílené paměti na jeden multiprocesor, a protože kernel využíval 8 KB sdílené paměti pro každý blok, mohlo současně

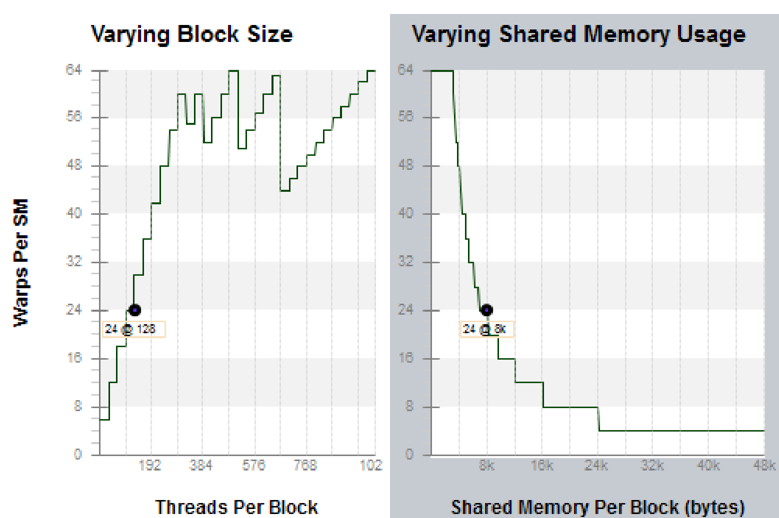
běžet maximálně 6 bloků z 16-ti možných. Na obrázku 14 jde vidět, že na 1 multiprocesor běželo pouze 24 warpů z 64. Obrázek 15 vlevo také značí další problém, kvůli kterému došlo ke ztrátě výkonu. Problém byl špatné nastavení velikosti bloků.



Occupancy Per SM

Active Blocks		6	16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Active Warps	23,34	24	64	0 6 12 18 24 30 36 42 48 54 60 64
Active Threads		768	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	36,5%	37,5%	100%	0% 25% 50% 75% 100%

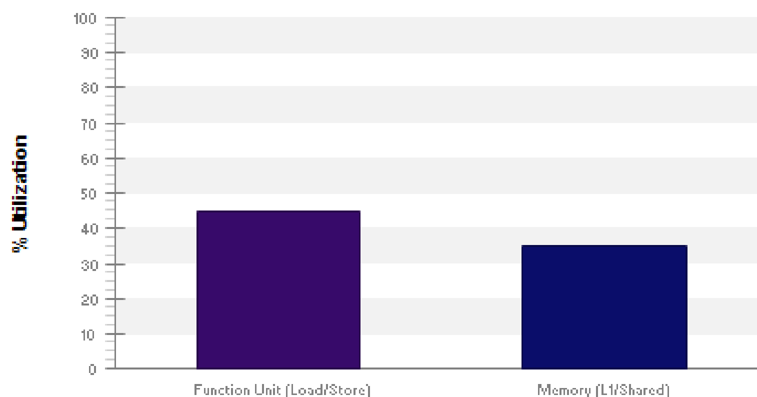
Obrázek 14: Obsazenost multiprocesoru na GeForce GT 630



Obrázek 15: Analýza latence - GeForce GT 630

Profilování na GeForce GTX 690

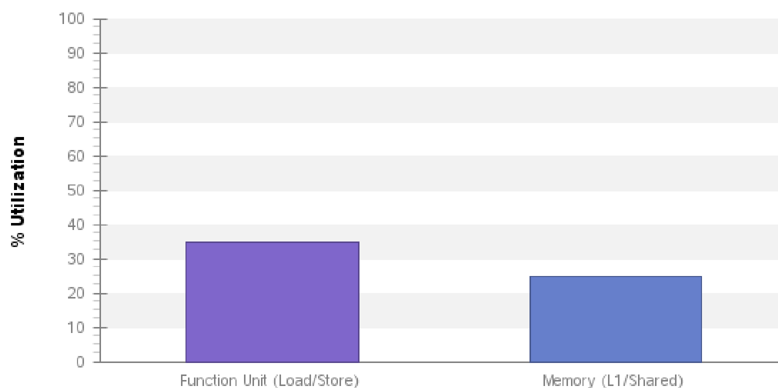
Při profilování této karty bylo zjištěno, že obsazenost multiprocesorů činila 33% a využití se pohybovalo okolo 40%. Tyto údaje lze vyčíst z obrázku 16. Při profilování algoritmu pracovalo 21 aktivních warpů na multiprocesor. Stejně jako u profilování karty GeForce GT 630 se projevil problém se sdílenou pamětí a se špatně nastavenou velikostí bloků.



Obrázek 16: Analýza latence - GeForce GTX 690

Profilování na Tesla K20XM

Dosažená obsazenost multiprocesoru byla pouhých 29,8%. Dle obrázku 17 je jasné, že využití karty se pohybovalo okolo 30%. Při spuštěného tohoto algoritmu na Tesla K20XM běželo pouze 19 warpů z 64 možných. Problém s nastavením sdílené paměti i s počtem vláken na jeden blok byl stejný jako v prvním případě u karty GeForce GT 630.



Obrázek 17: Využití karty Tesla K20XM

Řešení problému

Výsledek na obrázku 15 vyšel pro všechny grafické karty stejně. Profilování naznačilo, že řešením by mohlo být zvětšení velikosti bloků. Vzhledem k implicitně nastavené sdílené paměti o velikosti pole 1024 prvků, je z grafu patrné, že zlepšení výkonu by přineslo zvětšení bloků na 512 vláken. Pokud by byla zvolena varianta velikosti bloku 1024 vláken, musela by se z důvodu funkčnosti algoritmu také zvýšit velikost sdílené paměti. Sdílená paměť však má svou omezenou velikost, proto byla zvolena varianta velikosti bloku 512 vláken.

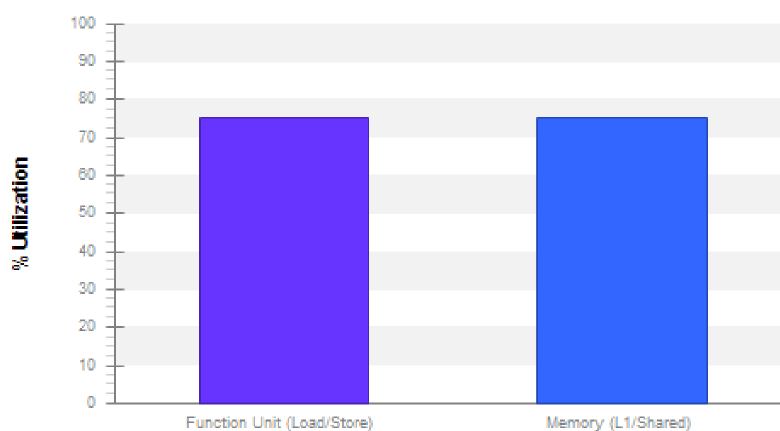
5.3.2 Profilování upraveného algoritmu

Algoritmus byl upraven a nyní ho bylo potřeba opět profilovat. To umožnilo zjistit, jestli došlo ke zlepšení výkonu. Profilování opět probíhalo pro všechny tři grafické karty a pro dvě varianty třídění. První varianta byla pro třídění 32 768 čísel. Druhou variantou bylo třídění pro 33 554 432 čísel.

Profilování na GeForce GT 630

Po úpravě algoritmu byla pro třídění 32 768 čísel dosažená obsazenost karty kernelem na 99,2%. Zlepšilo se také využití výkonu karty. Na obrázku 18 lze vidět razantní zlepšení výkonu oproti předchozí variantě na obrázku 13. Nyní se výsledný výkon pohyboval okolo 75%. Počet aktivních warpů na multiprocesor byl 64, což bylo také maximum.

Profilování pro třídění 33 554 432 čísel dopadlo naprosto podobně jako v prvním případě.

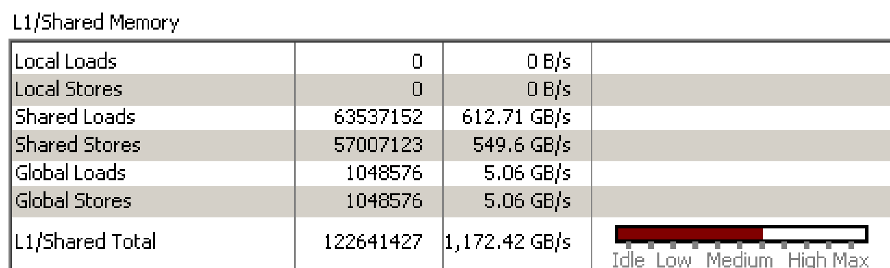


Obrázek 18: Využití karty GeForce GT 630 u upraveného algoritmu.

Profilování na GeForce GTX 690

Při třídění 32 768 čísel byla dosažená obsazenost karty byla 92% a celkové využití karty se pohybovalo okolo 75%. Počet aktivních warpů na multiprocesor byl na čísle 60.

Při třídění 33 554 432 čísel kleslo využití karty na 60%. Hlavním důvodem poklesu byl častý přístup do sdílené paměti. Obrázek 19 nám tento problém ilustruje.



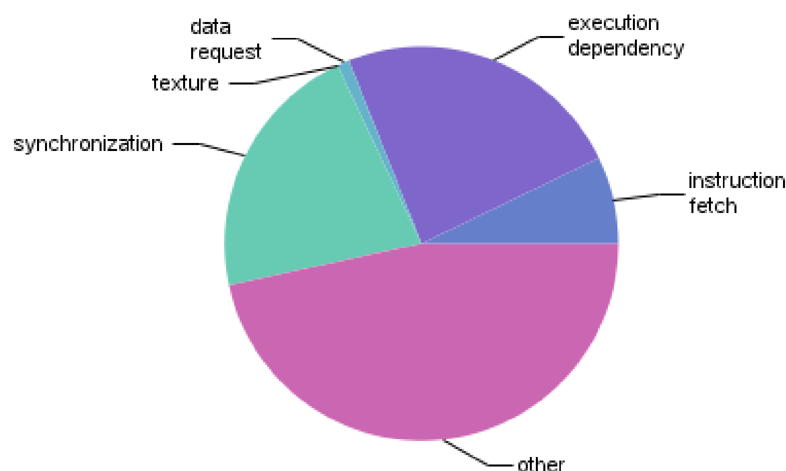
L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	63537152	612.71 GB/s	
Shared Stores	57007123	549.6 GB/s	
Global Loads	1048576	5.06 GB/s	
Global Stores	1048576	5.06 GB/s	
L1/Shared Total	122641427	1,172.42 GB/s	Idle Low Medium High Max

Obrázek 19: Problém s propustností paměti u GeForce GTX 690

Profilování na Tesla K20XM

Při profilování byla obsazenost multiprocesorů pouze 51% a celkový výkon se pohyboval okolo 45%. Hlavním důvodem byl takový, že pro třídění 32 768 čísel nebylo vytvořeno tolik bloků, aby se dostatečně využilo potenciálu této karty. Proto byl algoritmus profilován také pro 2 097 152 čísel, zda se potvrdí tato domněnka. Výkon se zvýšil na 55%. A obsazenost karty byla 99%. Pří třídění 33 554 432 čísel byl výkon totožný jako při třídění 2 097 152 čísel. Důvody, proč nebylo dosaženo většího výkonu, jsou znázorněny na obrázku 20, jedná se například o synchronizace vláken a závislostí spouštěných instrukcí. Jako u předchozí karty se objevil problém častého zápisu a čtení z paměti. Bohužel vzhledem k funkčnosti algoritmu tyto věci bohužel nejdou ovlivnit.



Obrázek 20: Využití karty Tesla K20XM u upraveného algoritmu.

Při pozdějším testování, pokud nebude řečeno jinak, bude vždy používán tento upravený algoritmus, který lze vidět na obrázku21.

```
1  int arrayLength = 1024;
2  uint   blockCount = N / arrayLength;        // 32 768 / 1024 = 32
3  uint  threadCount = arrayLength/2;
4  oddEvenMergeSortShared<<<blockCount, threadCount>>>(d_DstKey, d_DstVal, d_SrcKey,
        d_SrcVal, arrayLength, dir);
```

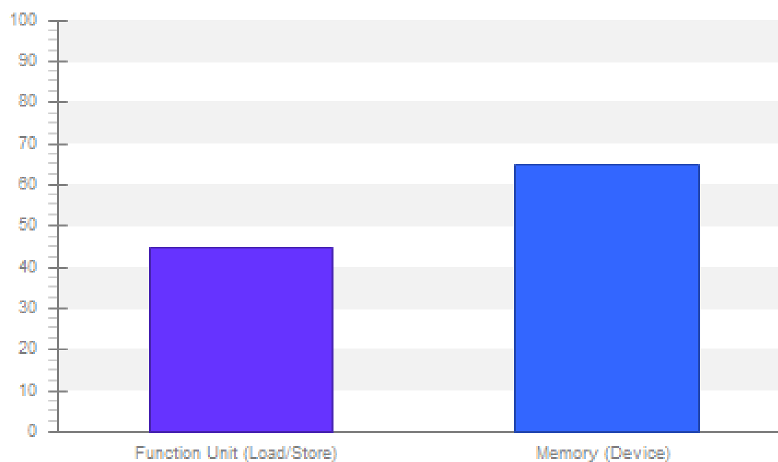
Obrázek 21: Spouštění upraveného kernelu se sdílenou pamětí

5.4 Profilování 2. fáze algoritmu

Kernel využívající globální paměť je spouštěn v cyklu a proto při běhu aplikace bylo vytvořeno několik desítek instancí tohoto kernelu. Spouštění probíhá s konstantním počtem vláken a bloků. Ve výchozím případě byla velikost bloku stanovena na 1024 vláken, přičemž celkově potřebujeme N/2 vláken, když N je počet prvků, které mají být setříděny. Profilování kernelu proběhlo opět pro třídění 32 768 čísel.

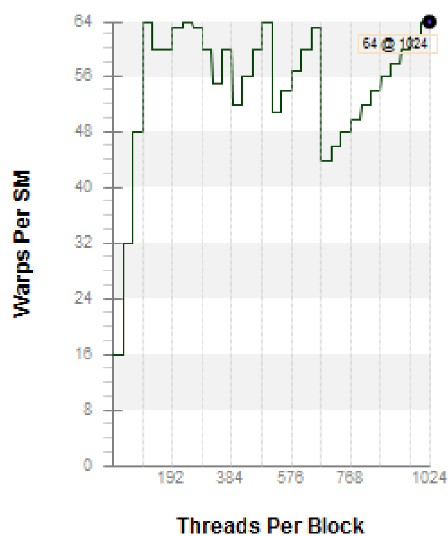
Profilování na GeForce GT 630

Obsazenost multiprocesoru se pohybovala nad 70%. Průměrný výkon je něco přes 50%. Obrázek 22 ukazuje, že výpočetní využití je mnohem nižší než využití paměti. Počet aktivní warpů se pohyboval okolo 45.



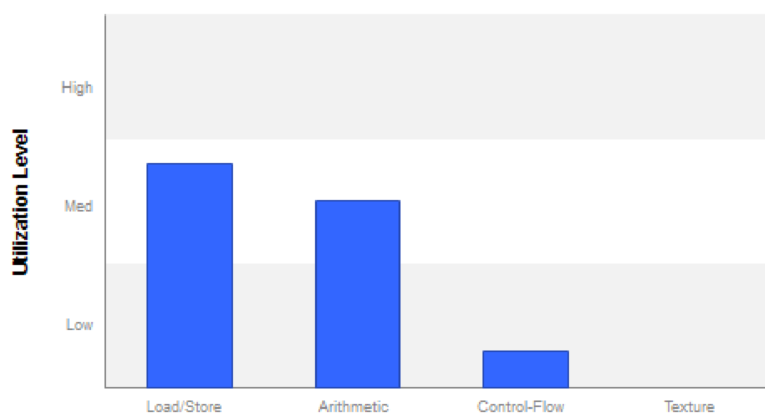
Obrázek 22: Využití karty GeForce GT 630 u 2.fáze algoritmu

Obrázek 23 zobrazuje možnosti velikostí bloků. Proto byla pro experimentování zvolena naprosto opačná varianta než velikost bloku 1024 vláken. Zvolena byla velikost 128 vláken.



Obrázek 23: Nabízené možnosti velikosti vláken

Výsledkem bylo, že průměrná obsazenost multiprocesoru všemi se pohybovala okolo 88%. Celkové využití karty vystoupalo nad 60%. Hlavním faktorem zlepšení bylo, že se zvýšil počet aktivních warpů na multiprocesor z 45 na 56. Výkon byl však stále limitován faktorem častého prací s daty v paměti. Neustále potřebné načítání a ukládání dat ubírá na výkonu, což dokládá také obrázek 24.

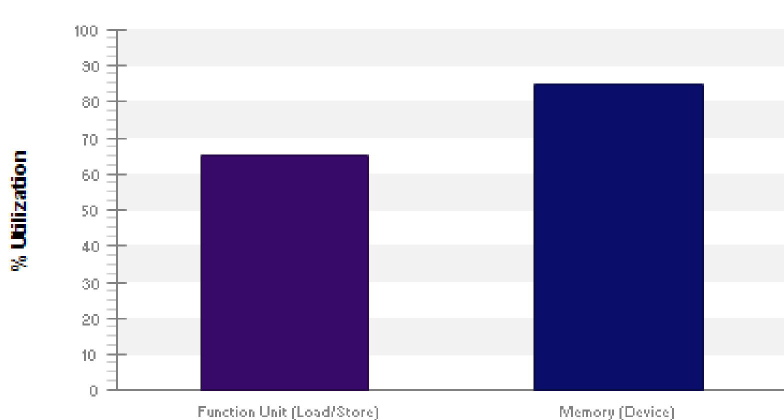


Obrázek 24: Graf zápisu a čtení

Výsledky pro GeForce GTX 690

Při profilování pro 32 768 čísel byla obsazenost multiprocesoru při velikosti bloku 1 024 vláken 76%, jelikož na multiprocesor bylo aktivních 49 warpů z 64 možných. Při profilování pro 2 097 152 čísel se obsazenost zvýšila na 80% a počet aktivních warpů na multiprocesor byl 51.

Pokud jsem změnil velikost bloků na 128 vláken a algoritmus se profiloval pro 2 097 152 čísel, tak se obsazenost karty vyšplhala na 85% a celkové využití karty, které je zobrazené na obrázku 25, bylo v průměru okolo 70%. Bohužel maximální výkon je limitován propustností paměti, protože časté čtení a zápis do paměti vykazují vysokou režii.



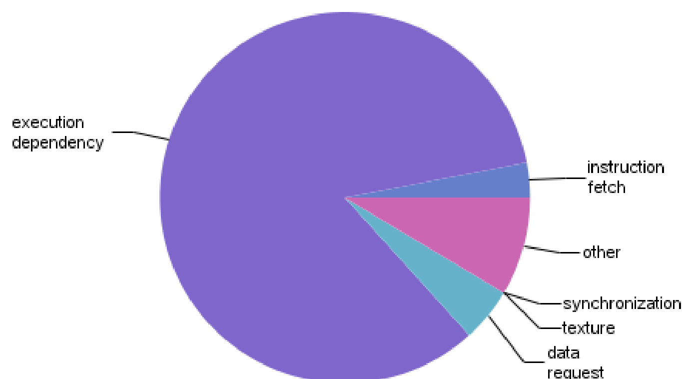
Obrázek 25: Využití karty GeForce GTX 690 u 2.fáze algoritmu

Výsledky pro Tesla K20XM

Celkový výkon se pohyboval nad 20% a obsazenost multiprocesoru okolo 50%. Hlavním důvodem byl nízký počet bloků, takže nebylo plně využito potenciálu grafické karty. Proto bylo profilování provedeno také pro 2 097 152 čísel. Při velikosti bloků 1 024 vláken se výkon pohyboval přibližně okolo 70%. Aktivních warpů na multiprocesor bylo 53.

Pokud se změnila velikost bloků na 128 vláken, tak se výsledný výkon příliš nezměnil, ale obsazenost multiprocesoru se přiblížila až k hranici 90%.

Bohužel 100% hranice opět nelze dosáhnout ze dvou důvodů. Tím prvním se stává šířka paměťového pásma. Příliš mnoho čtení a zápisu omezuje výkonnost jako při předešlých testech. Druhým problémem jsou závislosti mezi jednotlivými instrukcemi, kdy nemůže být spuštěna jiná instrukce, aniž by byla dokončena předešlá. Tento problém je znázorněn na obrázku 26. Toto je bohužel spojeno s daným algoritmem, kde nám tyto závislosti vytvářejí některé potřebné výpočty, na které musí čekat další instrukce.



Obrázek 26: Ukázka závislostí mezi spuštěním

5.5 Zhodnocení profilování

Při ladění bylo zjištěno špatné nastavení velikosti bloků. V prvním kernelu, který pracuje se sdílenou pamětí se podařilo dosáhnout výrazného zlepšení vytížení multiprocesorů. U třídění nízkého počtu čísel bylo zjištěno, že u silnějších grafických karet algoritmus nevyužije jejich maximálního výkonu. Problém způsobuje nízký počet bloků vzhledem k vyššímu počtu multiprocesorů na těchto kartách. Při vyšších číslech tříděných prvků se výkon zlepšuje, avšak zpomalení zde způsobují faktory synchronizace vláken a vzájemná závislost instrukcí. Kernel pracující s globální pamětí postihují stejné problémy, kromě synchronizace vláken. U prvního kernelu se sdílenou pamětí tedy byla zvolena velikost bloku 512 vláken. U kernelu druhého byla nastavena velikost bloku 128 vláken. Tato implementace bude součástí dalšího měření jak pro platformu CUDA tak také pro OpenCL.

6 Měření a porovnávání

V této kapitole se porovná výchozí algoritmus s upraveným algoritmem. Tyto výsledky se následně zhodnotí a zjistí se případné jiné problém nebo nedostatky algoritmu. Dalším krokem bude porovnání implementace bez sdílené paměti s upraveným algoritmem, který sdílenou paměť využívá. Nakonec se také porovnají platformy OpenCL a CUDA mezi sebou.

6.1 Porovnání výchozího a upraveného algoritmu

V první fázi bylo třeba porovnat výchozí a upravený algoritmus. Po provedeném profilování by měl být výsledek upraveného algoritmu lepší než výsledek výchozího. Po profilování a následném vylepšení algoritmu je více využíván výkon jednotlivých multiprocesorů na grafické kartě. Porovnání těchto dvou algoritmů proběhlo na platformě CUDA. Tabulka 2 obsahuje naměřené hodnoty výchozího algoritmu. V tabulce 3 jsou naměřené hodnoty upraveného algoritmu.

Počet čísel	**Čas (ms)**		
	GeForce GT 630	**GeForce GTX 690**	**Tesla K20XM**
4096	0,51	0,29	0,96
16 384	1,64	0,48	1,72
32 768	3,59	0,68	2,32
65 536	7,76	1,15	2,85
131 072	16,38	2,35	3,43
524 288	72,95	9,96	10,67
1 048 576	160,37	21,56	20,86
2 097 152	353,59	47,02	42,64
4 194 304	773,97	102,83	90,39
8 388 608	1 684,00	224,15	195,02
16 777 216	3 679,00	478,68	421,82
33 554 432	7 971,00	1 051,00	917,23
67 108 864	17 163,00	2 275,00	1 990,00
134 217 728	-	-	4 312,00
268 435 456	-	-	9 293,00

Tabulka 2: Hodnoty měření výchozího algoritmu v CUDA

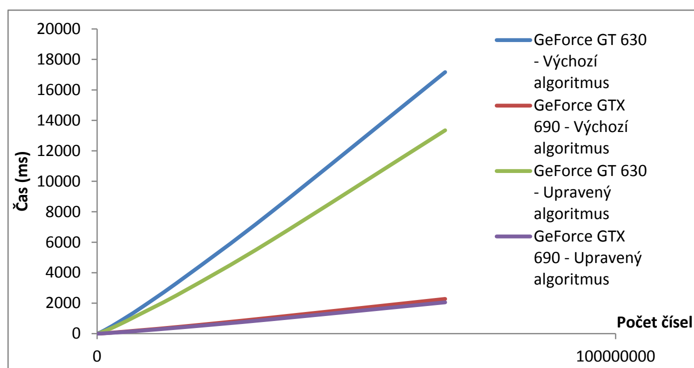
Počet čísel	Čas (ms)		
	GeForce GT 630	GeForce GTX 690	Tesla K20XM
4096	0,29	0,16	0,73
16 384	0,98	0,36	1,44
32 768	2,30	0,54	1,77
65 536	5,25	0,88	2,35
131 072	10,28	1,89	2,83
524 288	53,22	8,21	9,12
1 048 576	119,52	17,96	18,42
2 097 152	264,40	39,79	37,39
4 194 304	589,67	87,78	79,94
8 388 608	1 303,72	194,94	173,40
16 777 216	2 794,00	427,56	395,71
33 554 432	6 116,00	938,18	865,28
67 108 864	13 350,00	2 054,00	1 886,00
134 217 728	-	-	4 014,00
268 435 456	-	-	8 880,00

Tabulka 3: Hodnoty měření upraveného algoritmu v CUDA

Na grafu 27 můžeme vidět, že pro nejslabší kartu GeForce GT 630 byl upravený algoritmus téměř o 30% rychlejší než algoritmus výchozí. Pokud tyto dva algoritmy porovnáme na silnější kartě GeForce GTX 690, tak zjistíme, že s rostoucím počtem tříděných čísel klesá zrychlení.

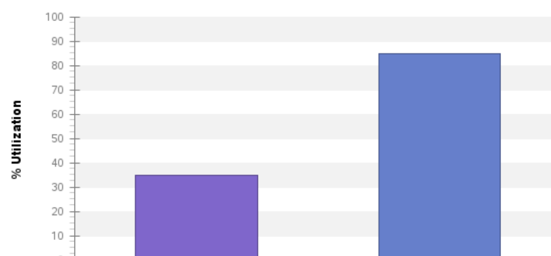
Kernel prováděný první fází algoritmu má i pro 33 554 432 tříděných čísel stále stejný výkon. Problém nastává u kernelu v 2.fáze algoritmu, který využívá pouze globální paměť.

Tento problém je vidět u silnější karty GeForce GTX 690, kde nám s rostoucím počtem tříděných čísel klesalo zrychlení.



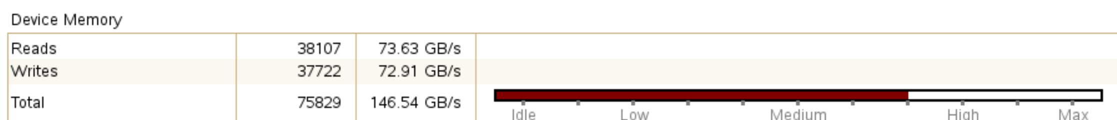
Obrázek 27: Porovnání algoritmů na platformě CUDA

Proto jsem zkusil profilovat algoritmus pro 33 554 432 čísel na nejsilnější grafické kartě Tesla K20XM. Obsazenost multiprocesorů se pohybovala okolo 90%. Avšak jak je patrné z obrázku 28, využití této karty bylo nízké.



Obrázek 28: Využití upraveného algoritmu na Tesla K20XM pro 33 554 432 čísel

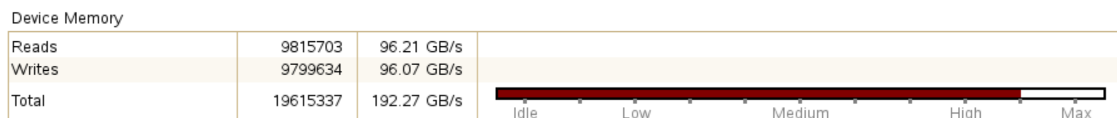
Porovnal jsem tedy také paměťové operace pro grafickou kartu Tesla K20XM. Při třídění 131 072 čísel tvořilo čtení a zápis do paměti dohromady přenos 146,54 GB za sekundu. Při třídění 33 554 432 čísel byl tento přenos 192,27 GB za sekundu.



Device Memory

Reads	38107	73.63 GB/s	
Writes	37722	72.91 GB/s	
Total	75829	146.54 GB/s	Idle Low Medium High Max

Obrázek 29: Paměťová režie Tesla K20XM pro 131 072 čísel



Device Memory

Reads	9815703	96.21 GB/s	
Writes	9799634	96.07 GB/s	
Total	19615337	192.27 GB/s	Idle Low Medium High Max

Obrázek 30: Paměťová režie Tesla K20XM pro 33 554 432 čísel

Pokud porovnáme změřené hodnoty výchozího algoritmu z tabulky 2 s hodnotami upraveného algoritmu v tabulce 3, můžeme usoudit, že profilování algoritmu patří k důležitým procesům při vývoji aplikace běžící na grafické kartě. Můžeme vidět, že na nejsilnější kartě Tesla K20XM bylo třídění pro 268 435 456 čísel rychlejší o půl vteřiny. U nejslabší karty GeForce GT 630 byl tento rozdíl razantnější, protože při třídění 67 108 864 čísel se vytvořil rozdíl mezi upraveným a neupraveným algoritmem téměř 4 vteřiny.

Bylo však zjištěno, že silnější grafické karty zpomaluje 2. fáze algoritmu, kde kernel pracuje pouze s globální pamětí. Vyšší počet paralelních procesů u těchto grafických karet také vyžaduje větší režii zápisu a čtení z globální paměti. To je však limitováno paměťovou propustností, která snížila výkon. Optimalizace tohoto algoritmu tedy nebyla optimální pro třídění čísel v řádu desítek a stovek miliónů.

Bohužel pro grafické karty GeForce GT 630 a GeForce GTX 690 bylo možno provádět třídění maximálně pro 67 108 864 čísel. Problém nastal při alokování paměti na grafické kartě, jelikož v globální paměti už nebyla dostatečná kapacita. U grafické karty Tesla K20XM bylo možno provádět třídění až pro 268 miliónů čísel díky tomu, že má větší velikost globální paměti, než předchozí dvě karty. Porovnání velikostí globálních pamětí mezi jednotlivými kartami lze vyčíst z tabulky 1.

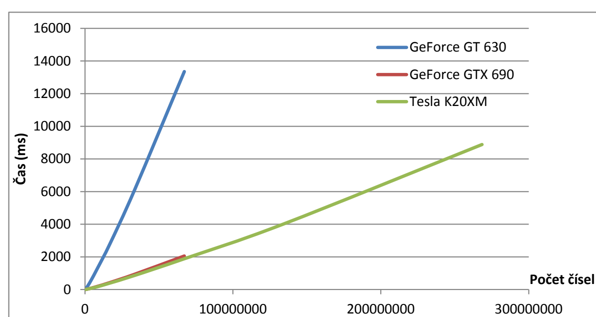
6.2 Porovnání grafických karet

Porovnávání rychlosti jednotlivých grafických karet probíhalo s upraveným (rychlejším) algoritmem, jehož naměřené hodnoty jsou znázorněny v tabulce 3.

Grafická karta GeForce GT 630 má pouze 1 multiprocesor, který v sobě nese 192 CUDA jader. Karta GeForce GTX 690 má 8 multiprocesorů, kde každé z nich má 192 CUDA jader. Výkon druhé zmíněný karty by tedy měl být podstatně vyšší, což se také potvrdilo a jde vidět na obrázku 31. Dalo by se taky očekávat, že výsledný výkon by mohl být až 8 x vyšší. Výsledek měření však ukázal, že vyšší výkon byl přibližně 6,7 x vyšší. Tyto údaje zobrazuje tabulka 3. Příčinou této ztráty je, že obsazenost multiprocesoru u obou kernelů se blížila hranici 90%, ale těch zbývajících 10% se projevilo na mírné ztrátě výkonu.

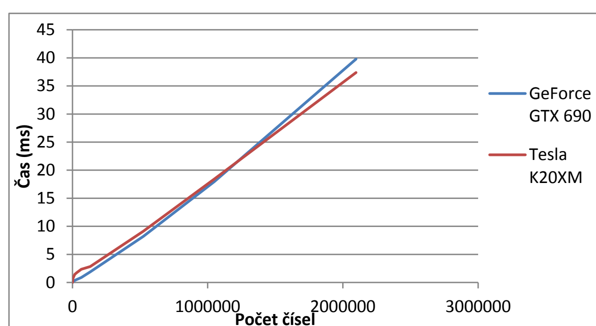
Testování proběhlo také na školním databázovém serveru zvaným CODD na grafické kartě Tesla K20XM. Tato grafická karta má vestavěných 14 multiprocesorů s celkem 2 688 CUDA jádry. Výsledky měření na této kartě by podle předpokladu měly být nejlepší, avšak podle výsledků měření v tabulce 3, bylo zjištěno, že lepší výkon se projevil až od 2 miliónů čísel a výše. Právě u této hranice se pohybovala obsazenost multiprocesoru těsně pod 90%. Bohužel 100% hranice opět nelze dosáhnout ze dvou důvodů. Tím prvním se stala šířka paměťového pásma. Příliš mnoho čtení a zápisu omezovalo výkonnost jako při předešlých testech. Druhým problémem byla závislosti mezi jednotlivými instrukcemi,

kdy nemohla být spuštěna jiná instrukce, aniž by byla dokončena instrukce předešlá. Toto bylo bohužel spojeno s daným algoritmem, kde byly tyto závislosti vytvořeny některými potřebnými výpočty, na které musely čekat další instrukce.



Obrázek 31: Porovnání karet - CUDA

Nyní bylo potřeba zjistit, proč byl u karty Tesla K20XM horší čas než u karty GeForce GTX 690 do 2 miliónů čísel (vyjádřeno na obrázku 32). Hlavním problémem bylo špatné využití jednotlivých multiprocesorů. po analýze algoritmu na této kartě, se zjistilo, že docházelo ke větším ztrátám na výkonu pro nízký počet tříděných čísel. Hlavním důvodem byl nízký počet bloků a karta nevyužívala svého potenciálu. U lepší karty Tesla K20XM tyto ztráty na výkonu pro nižší počet čísel byly vyšší než u grafické karty GeForce GTX 690.



Obrázek 32: Bližší pohled srovnání GeForce GTX690 a Tesla K20XM

6.3 Měření algoritmu bez sdílené paměti

Jak už bylo napsáno v teoretické části, sdílená paměť je mnohem rychlejší než paměť globální. Proto proběhlo testování, aby se zjistilo, jak velkou roli hrála sdílená paměť v tomto algoritmu. Původní algoritmus využíval kombinace sdílené i globální paměti. Nyní byl vytvořen algoritmus bez sdílené paměti, který pracoval pouze s globální pamětí. Výsledky pro platformu OpenCL jsou v tabulce 5. Tabulka 4 zobrazuje výsledky měření pro platformu CUDA.

Počet čísel	**Čas (ms)**		
	GeForce GT 630	**GeForce GTX 690**	**Tesla K20XM**
4096	0,59	0,43	1,47
16 384	1,34	0,66	1,99
32 768	2,50	0,88	2,40
65 536	6,49	1,27	2,91
131 072	13,53	2,6	3,75
524 288	64,69	10,42	11,73
1 048 576	141,38	22,17	22,19
2 097 152	310,88	47,93	45,04
4 194 304	678,28	104,06	94,67
8 388 608	1 481,60	227,28	202,05
16 777 216	3 216,00	492,89	436,54
33 554 432	6 958,00	1 067,00	942,14
67 108 864	15 046,00	2 315,00	2 041,00
134 217 728	-	-	4 409,00

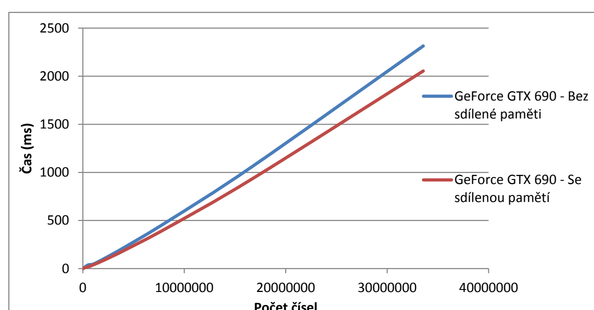
Tabulka 4: Hodnoty měření upraveného algoritmu bez sdílené paměti - CUDA

Počet čísel	Čas (ms)	
	GeForce GT 630	GeForce GTX 690
4096	0,71	0,65
16 384	1,57	0,93
32 768	3,13	1,19
65 536	7,19	1,87
131 072	16,32	3,11
524 288	69,48	11,46
1 048 576	150,95	22,98
2 097 152	331,14	48,72
4 194 304	718,21	104,93
8 388 608	1 569,51	227,53
16 777 216	3 398,00	492,99
33 554 432	7 335,00	1 068,00
67 108 864	15 838,00	2 316,00

Tabulka 5: Hodnoty měření upraveného algoritmu bez sdílené paměti - OpenCL

Na obrázku 33 jde vidět porovnání implementace algoritmu se sdílenou pamětí a bez ní na grafické kartě GeForce GTX 690. Rozdílní výsledek se dal čekat obrovský. Sdílená paměť, která byla používaná v první fázi algoritmu, však neprováděla tolik operací z celkového počtu práce. Jak již bylo zmíněno dříve, poměr prvního a druhého kernelu byl při počtu tříděných číslech v řádech miliónů v poměru 5:95. Proto, když jsme nahradili 5% algoritmu, který využíval sdílenou paměť, nedalo se očekávat, že výkon rapidně naroste.

Výsledkem tohoto měření však bylo, že sdílená paměť v určité míře algoritmus zrychlovala.



Obrázek 33: Porovnání algoritmů se sdílenou pamětí a bez ní.

7 Porovnání OpenCL a CUDA

Dalším důležitým měřením bylo porovnat obě platformy OpenCL a CUDA mezi sebou. Tato část se bude zabývat porovnáváním z hlediska času. V prvním kroku se změřila celková doba provádění všech instancí kernelů. Pro toto porovnávání byl použitý upravený algoritmus. Porovnávání obou platforem bylo změřeno pouze na grafických kartách GeForce GT 630 a GeForce GTX 690. Bohužel na grafické kartě Tesla K20XM nebyly potřebné nástroje pro zprovoznění aplikace implementované na platformě OpenCL.

7.1 Měření kernelů

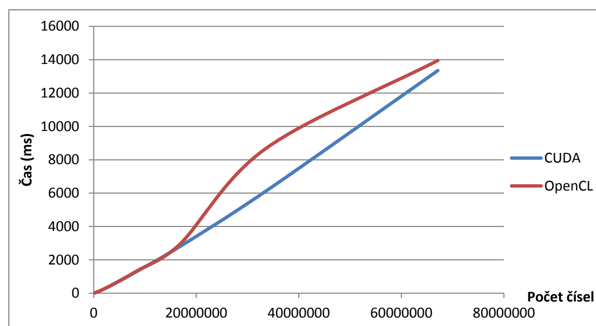
Celkový čas je součet trvání všech kernelů. Tento čas vystihuje pouze dobu běhu kernelu. V tomto čase nejsou započítány alokace paměti, přenos dat na grafickou kartu ani žádné jiné procesy.

Počet čísel	Čas (ms)			
	GT 630 CUDA	OpenCL	GTX690 CUDA	OpenCL
4096	0,29	0,32	0,16	0,51
16 384	0,98	1,04	0,36	0,81
32 768	2,30	2,31	0,54	1,01
65 536	5,25	5,11	0,88	1,52
131 072	10,28	11,12	1,89	2,17
524 288	53,22	54,55	8,21	8,67
1 048 576	119,52	122,38	17,96	18,40
2 097 152	264,40	272,13	39,79	40,09
4 194 304	589,67	604,77	87,78	88,14
8 388 608	1 303,00	1333,00	194,94	194,55
16 777 216	2 794,00	2 931,00	427,56	428,94
33 554 432	6 116,00	8 672,00	938,18	940,06
67 108 864	13 350,00	13 953,00	2 054,00	2 056

Tabulka 6: Doba trvání kernelů - CUDA a OpenCL.

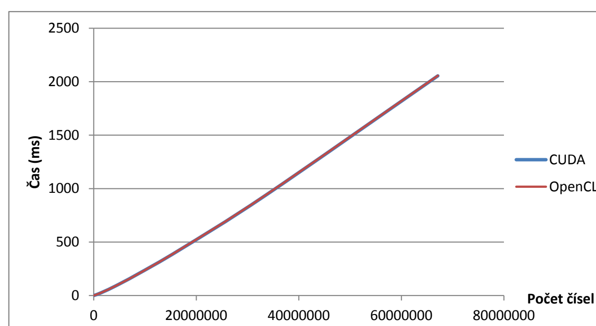
Z tabulky 6 je zřejmé, že kernely v OpenCL trvaly o něco déle než v CUDA. Avšak nejednalo se o příliš velké rozdíly. U karty GeForce GT 630 se při třídění 67 108 864 čísel projevil rozdíl půl vteřiny. U druhé karty GeForce GTX 690 byl taky čas kernelů mírně horší na platformě OpenCL, avšak pro třídění 67 108 864 čísel se to projevilo pouze v rámci milisekund.

Na grafu 35 můžeme vidět, že kernely v OpenCL trvaly delší dobu, ovšem časové výchylky nebyly konstantní. Toto mohlo zapříčinit, že optimalizace aplikace OpenCL byla horší než optimalizace aplikace na platformě CUDA.



Obrázek 34: Srovnání CUDA a OpenCL na kartě GeForce GT 630

Na obrázku 35 jde vidět srovnání těchto dvou platforem na grafické kartě GeFroce GTX 690. Na obrázku je jasně vidět, že doba trvání kernelů na platformě OpenCL se maximálně přiblížila době trvání kernelů na platformě CUDA.



Obrázek 35: Srovnání CUDA a OpenCL na kartě GeForce GTX 690

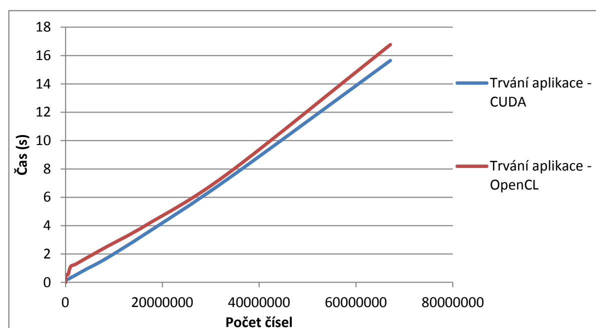
Z těchto měření jde usoudit, že algoritmus vytvořený v OpenCL měl na grafické kartě GeForce GT 630 delší trvání. Naopak na kartě GeForce GTX 690 se jeho neměřené hodnoty téměř blížily hodnotám naměřeným na platformě CUDA.

7.2 Měření trvání celé aplikace

Druhým krokem srovnání bylo měření, jak dlouho trvala celá aplikace. Toto měření proběhlo na grafické kartě GeForce GT 630. Podle výsledků v tabulce 7 se dá usoudit, že aplikace na platformě OpenCL trvají delší dobu. Toto tvrzení také dokazuje obrázek 36. Bylo třeba určit slabá místa aplikace OpenCL. Mezi tyto slabá místa platformy OpenCL nezvratně patří například vytváření kontextu, fronty, vytváření objektu kernelu, vytváření spustitelného programu a další nezbytné akce pro úspěšné spuštění OpenCL aplikace. Tyto procedury podstatně zpomalují celou aplikaci.

Počet čísel	Čas (s)	
	CUDA	OpenCL
4096	0,15	0,45
16 384	0,16	0,48
32 768	0,16	0,49
65 536	0,16	0,49
131 072	0,19	0,51
524 288	0,22	0,56
1 048 576	0,32	1,11
2 097 152	0,51	1,28
4 194 304	0,91	1,69
8 388 608	1,68	2,49
16 777 216	3,48	4,06
33 554 432	7,30	7,68
67 108 864	15,65	16,77

Tabulka 7: Doba trvání celé aplikace - CUDA a OpenCL.



Obrázek 36: Doba trvání aplikací - CUDA a OpenCL.

7.3 Rozdíly v implementaci

Vytváření aplikace na platformě OpenCL přináší potřebu implementovat více objektů než v případě aplikace na platformě CUDA. Níže budou stručně budou popsány tyto důležité implementace v technologii OpenCL, které programátor nemusí při použití platformy CUDA řešit. Tyto OpenCL funkce je nutné vykonat před spuštěním kernelu a budou popsány pouze nejdůležitější parametry těchto funkcí.

1. **Výběr platformy**
   První fáze spočívá ve výběru platformy.
   `clGetPlatformIDs(1,&platformID,NULL)`

2. **Získání identifikátoru zařízení**
   Platforma může poskytovat jedno nebo více zařízení. První parametr je ID platformy, druhým parametrem udáváme typ zařízení, které chceme použít. Třetí parametr značí kolik identifikátorů zařízení chceme vrátit a ten se nám vrátí do čtvrtého předaného parametru.
   `clGetDeviceIDs(platformID,CL_DEVICE_TYPE_GPU,1,&device_id,NULL)`

3. **Vytvoření kontextu**
   V další fázi je potřeba vytvořit kontext, který může zahrnovat jedno nebo více zařízení. V příkladu níže bude kontext zahrnovat jedno zařízení, což je dáno druhým parametrem. Třetí parametr je ID zařízení.
   `context=clCreateContext(0,1,&device_id,NULL,NULL,&errorCode)`

4. **Vytvoření fronty příkazů**
   Dalším krokem je vytvoření fronty příkazů. První parametr udává kontext, ve kterém bude tato fronta vytvořena. Druhý parametr je identifikátor zařízení, ve kterém se budou spouštět příkazy uložené ve frontě. Třetí parametr nastavuje požadované vlastnosti fronty, v našem případě je zde povoleno profilování. Toto profilování je potřeba mít povoleno, aby jsme později mohli měřit výkon kernelů pomocí událostí.
   ```
   commandQueue=clCreateCommandQueue(context,device_id,
   CL_QUEUE_PROFILING_ENABLE,&errorCode)
   ```

5. **Načtení kernelu**
   Nyní je potřeba načíst zdrojový kód kernelu. Kernel lze definovat přímo v programu jako řetězec, nebo je nutno kód kernelu načíst ze souboru. Načítání kernelu ze souboru může být ve formě zdrojového kódu nebo v binární podobě.

6. **Vytvoření programu**
   Po načtení kernelu je potřeba vytvořit objekt pro uložení programu. První parametr obsahuje odkaz na kontext a druhý počet ukazatelů na řetězec v poli třetího parametru. Třetí parametr je pole řetězců obsahující zdrojový kód kernelu.
   ```
   cl_program program=clCreateProgramWithSource(context,1,
   (const char**) &source_str, NULL, NULL)
   ```

7. **Vytvoření spustitelné verze programu**
   Z vytvořeného objektu programu, který je předán jako první parametr, je nyní potřeba vytvořit spustitelnou verzi.
   `clBuildProgram(program,0,NULL,NULL,NULL,NULL)`

8. **Vytvoření objektu kernelu**
   Spustitelná verze programu může obsahovat více kernelů. Pro každý kernel, který bude spouštěn, je nutné vytvořit jeden objekt kernelu. Prvním parametrem je předáván spustitelný objekt programu. Druhý parametr je název kernelu, pro který chceme vytvořit objekt kernelu.
   ```
   cl_kernel kernelObject=clCreateKernel(programObject,
   ”JmenoKernelu”,&errorCode)
   ```

Společnou fází obou platforem je alokování paměti na grafické kartě a následné překopírování dat. Tento proces probíhá na platformě CUDA i OpenCL. Proces alokování paměti a kopírování dat je téměř totožný. U technologie OpenCL se to však liší tím, že při alokování paměti máme možnost definovat, zda tato část paměti bude jen pro zápis, čtení nebo obojí.

Další rozdíl spočívá v tom, že na platformě OpenCL je nutné nastavení parametrů kernelu ještě před jeho spuštěním. Zatímco na platformě CUDA se tyto parametry pohodlně předávají vždy při volání kernelu, u technologie OpenCL je nutné nastavit tyto argumenty každému kernelu zvlášť ještě před jeho spuštěním.

Rozdílnou implementací se také stalo definování funkce zařízení. Zatímco na platformě CUDA jsme mohli definovat funkci zařízení pomocí `__device__`,kterou později bylo možné volat z kernelu, tak platforma OpenCL tuto možnost nenabízí.

Volání kernelu na platformě CUDA

Zde můžeme vidět příklad volání kernelu na platformě CUDA. V ostrých závorkách se udává velikost mřížky. První parametr `blockCount` značí počet bloků v mřížce a druhý parametr `threadCount` nastavuje počet vláken pro každý blok. Na konci v kulatých závorkách jsou předávané parametry.

```
Kernel<<<blockCount, threadCount>>>(argument1,argument2)
```

Volání kernelu na platformě OpenCL

Volání kernelu s argumenty je na platformě OpenCL složitější. Jako první krok je důležité parametry pro daný kernel nastavit. A až poté se volá kernel. Způsob vytváření mřížky je zde oproti technologii CUDA odlišný. Parametr `local` značí počet vláken pro každou pracovní skupinu. Druhý parametr `global` nastavuje celkový počet vláken pro celou mřížku. Takže technologie OpenCL si počet pracovních skupin spočítá sama z těchto dvou hodnot.

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_OutputKey)
clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_OutputVal)
clEnqueueNDRangeKernel(commands,kernel, 1, NULL, &global,
&local, 0, NULL, NULL)
```

7.4 Zhodnocení rozdílů

Z výsledků měření vyplynulo, že tyto dvě architektury jsou si velice podobné. Technologie OpenCL se výkonnostně výrazně přiblížila výkonu technologie CUDA. U této technologie vzniká ovšem malý časový rozdíl, který způsobují potřebné programátorské činnosti k vytvoření a spuštění aplikace. Tyto činnosti programátor technologie CUDA nemusí řešit a přináší to značnou výhodu. Na druhou stranu tyto nezbytné činnosti v technologii OpenCL přináší výhodou, že tato technologie není závislá na jednom výrobci grafické karty, její kód je přenositelný na více zařízení jiných výrobců. Další výhodou OpenCL může být, že se jedná o otevřený standard. Nevýhodou OpenCL může být, že pokud je její kód optimalizovaný na grafickou kartu jednoho výrobce, není řečeno, že na grafické kartě vyrobené jiným výrobcem poběží kód stejně efektivně. Jako hlavní nevýhoda technologie CUDA je její závislost na grafických kartách výrobce NVIDIA.

8 Závěr

Cílem této bakalářské práce bylo zjistit, zda jde využít výkon moderních grafických karet také pro jiné účely, než je zpracování 2D a 3D obrazu. Pro toto zjišťování byly zvoleny řadící algoritmy, což je jeden z často se vyskytujících problémů.

Pro provádění těchto algoritmů pomocí grafické karty byly zvoleny technologie CUDA a OpenCL, které byly jednotlivě představeny v teoretické části práce. Pro technologii OpenCL byla představena architektura této platformy a následné popisy jednotlivých modelů. Kapitola věnující se technologii CUDA popsala rozdíl mezi grafickou kartou a procesorem, architekturu této platformy a typy pamětí. Je třeba si uvědomit, že teoretická část nemohla dodat kompletní přehled o obou porovnávaných technologiích. Tyto technologie nabízejí tolik možností, že jejich popis je pouze stručné představení do této problematiky. Jako řadící algoritmus byl vybrán Odd-Even merge sort, který byl navržen právě pro práce na paralelních systémech.

Druhá polovina práce zahrnovala teoretický rozbor algoritmu a popis jeho implementace. Byly zde popsány také rozdíly, které bylo nutné implementovat navíc na platformě OpenCL oproti implementaci na platformě CUDA. Významnou částí práce byla analýza a optimalizace tohoto algoritmu k zajištění maximálního výkonu a maximálního využití grafické karty. Po optimalizaci došlo k testování na několika grafických kartách a srovnávání výkonu před a po optimalizaci. Na závěr se provedlo srovnání obou technologií. Toto srovnání proběhlo jak pro dobu trvání kernelů, tak pro běh celé aplikace. Z výsledků srovnávání vyplynulo, že co se týče výkonu i architektury, jsou si obě technologie velice blízké.

Výhodou technologie OpenCL bez pochyb je nezávislost na výrobci grafické karty. Tato technologie se dá použít na jakémkoliv vícejádrovém heterogenním systému. Lehkou nevýhodou se stává programovací část, která zahrnuje nutnost vytvořit několik objektů, aby bylo umožněno spustit kernel. Nevýhodou technologie CUDA je, že je striktně vázána na grafické karty firmy NVIDIA. Avšak výsledný čas na těchto kartách byl o něco lepší než pro technologii OpenCL.

Budoucí rozšíření práce by mohlo zahrnovat implementaci a testování nějakého dalšího paralelního třídícího algoritmu. Případně by se tato práce mohla rozšířit o další použitou technologii. Například by se mohlo jednat o technologii ATI Stream, která také slouží pro paralelní zpracování úloh na grafických kartách.

9 Reference

[1] David B. Kirk, Wen-mei W. Hwu, *Programming massively parallel processors*. Burlington: Elsevier, 2010.

[2] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, Dan Ginsburg: *OpenCL programming guide*. Michigan: Edwards Brothers, 2011.

[3] Wen-mei W. Hwu, *GPU Computing Gems*. Burlington: Elsevier, 2011.

[4] Shane Cook, *CUDA Programming - A Developer's Guide to Parallel Computing with GPUs*. Waltham: Elsevier, 2013.

[5] Khronos OpenCL Working Group: The OpenCL Specification, [online]. [cit. 2014-04-29].
Dostupné z: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[6] Cliff Woolley: Introduction to OpenCL, [online]. [cit. 2014-04-29].
Dostupné z: http://www.cc.gatech.edu/˜vetter/keeneland/tutorial-2011-04-14/06-intro˙to˙opencl.pdf

[7] NVIDIA: CUDA C BEST PRACTICES GUIDE, [online]. [cit. 2014-04-29].
Dostupné z: http://docs.nvidia.com/cuda/pdf/CUDA˙C˙Best˙Practices˙Guide.pdf

[8] NVIDIA: CUDA C PROGRAMMING GUIDE, [online]. [cit. 2014-04-29].
Dostupné z: http://docs.nvidia.com/cuda/pdf/CUDA˙C˙Programming˙Guide.pdf

[9] Mark Harris, Michael Garland: Designing Efficient Sorting Algorithms for Manycore GPUs, [online].[cit. 2014-04-29].
Dostupné z: http://www.nvidia.com/docs/io/67073/nvr-2008-001.pdf

[10] Erik Sintorn: Fast Parallel GPU-Sorting Using a Hybrid Algorithm, [online]. [cit. 2014-04-29].
Dostupné z: http://www.cse.chalmers.se/˜uffe/hybridsort.pdf

[11] Carl Burch, Hendrix College: Introduction to parallel and distributed algorithms , [online]. [cit. 2014-04-29].
Dostupné z: http://www.toves.org/books/distalg/distalg.pdf

[12] Matt Pharr, Randima Fernando, *GPU gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Boston: Addison-Wesley, 2005.

[13] Apple: OpenCL Programming Guide for Mac , [online]. [cit. 2014-04-29].
Dostupné z: https://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL˙MacProgGuide/OpenCLLionMemoryObjects/OpenCLLionMemoryObjects.html

[14] Jeremiah van Oosten: Introduction to CUDA 5.0, [online]. [cit. 2014-04-29].
Dostupné z: http://3dgep.com/?p=4151

[15] Wikipedia - The Free Encyclopedia: Batcher odd–even mergesort, [online].
[cit. 2014-05-05].
Dostupné z: http://en.wikipedia.org/w/index.php?title=Batcher˙odd%E2%80%93even˙mergesort&oldid=590851639

[16] Carl Burch, Hendrix College: Introduction to parallel and distributed algorithms, [online]. [cit. 2014-04-29].
Dostupné z: http://www.toves.org/books/distalg/distalg.pdf

A Obsah CD

A.1 Technická zpráva

Technická zpráva se nachází v adresáři **TechnickaZprava/** pod názvem lin0038.pdf.

A.2 Zdrojové kódy

Zdrojové kódy pro platformy CUDA a OpenCL se nachází v adresáři **ZdrojoveKody/**.